

### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF







# Qt 5.9

## C++ 开发指南

王维波 栗宝鹃 侯春望 ◎ 著



人民邮电出版社  
北京





## 图书在版编目 (C I P) 数据

Qt 5.9 C++开发指南 / 王维波, 栗宝鹃, 侯春望著

— 北京 : 人民邮电出版社, 2018.5

ISBN 978-7-115-47868-9

I. ①Q… II. ①王… ②栗… ③侯… III. ①C++语言—程序设计—指南 IV. ①TP312.8-62

中国版本图书馆CIP数据核字(2018)第039642号

## 内 容 提 要

本书以 Qt 5.9 LTS 版本为开发平台, 详细介绍了 Qt C++开发应用程序的技术, 包括 Qt 应用程序的基本架构、信号与槽工作机制、图形显示的 Graphics/View 架构、数据编辑和显示的 Model/View 架构、对话框和多窗口的设计与调用方法等, 介绍了常用界面组件、文件读写、绘图、图表、数据可视化、数据库、多线程、网络和多媒体等模块的使用。每个编程主题都精心设计了完整的实例程序。

通过阅读本书, 可了解 Qt C++开发应用程序所需的基本技术。本书适合具有 C++语言编程基础, 希望应用 Qt C++开发跨平台应用程序的读者阅读。

- 
- ◆ 著 王维波 栗宝鹃 侯春望  
责任编辑 杨大可  
责任印制 焦志炜
  - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号  
邮编 100164 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
大厂聚鑫印刷有限责任公司印刷
  - ◆ 开本: 800×1 000 1/16  
印张: 29.5  
字数: 696 千字 2018 年 5 月第 1 版  
印数: 1-2 400 册 2018 年 5 月河北第 1 次印刷
- 

定价: 89.00 元

读者服务热线: (010)81055410 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京东工商广登字 20170147 号



# 前言

从 1994 年至今, Qt 已经走过了 20 多年的发展历程, 日趋成熟并广受欢迎。Qt 是非常优秀的跨平台应用开发框架, 在专业应用领域, 很多应用软件都是采用 Qt C++ 开发的, 比如在地球物理研究领域, 开发的专业软件一般需要在 Linux、Windows 或 macOS 等多种平台上运行, 使用 Qt 就是非常好的选择。

本书主要介绍如何使用 Qt 进行 C++ 应用程序开发。C++ 是使用最广泛的编程语言之一, 在各个专业领域有很多大型类库都是用 C++ 编写的, 在不同的专业研究领域可以找到很多开源的用 C++ 编写的类库或算法程序, 这对于专业软件开发是非常有利的。所以, 扎实地掌握 Qt C++ 编程如同掌握了一件利器, 无论是通过编程实现自己的专业研究成果, 还是从事专业的软件开发都是有长远意义的。

本书以 Qt 5.9 LTS (Long Term Supported) 版本为例介绍 Qt C++ 开发, 所有实例程序在 Windows 7 平台上测试, 需要使用 MSVC 编译器的时候使用 Visual Studio 2015。由于 Qt 的跨平台特性, 这些实例程序在 Linux 和 macOS 系统上基本上无需修改就可以编译, 所以, 本书介绍的内容也适用于 Linux 和 macOS 平台上的应用程序开发。

本书不对 C++ 语言的基本特性作介绍, 需要读者已经掌握了 C++ 语言编程的基本原理, 对类的概念和使用比较熟悉。如果对 C++ 语言还不够熟悉, 可以找一本专门介绍 C++ 语言的书学习, 有很多很好的专门介绍 C++ 语言的书。

Qt 实际上是一套应用程序开发类库, Qt 类库由许多模块组成, 如核心的 GUI 组件模块 Qt Widget, 用于数据库访问的 Qt SQL 模块, 用于二维图表显示的 Qt Charts 模块, 用于数据三维显示的 Qt Data Visualization 模块, 用于网络编程的 Qt Network 模块等。Qt 的模块很多, 在一本书里很难全面介绍到, 本书只介绍应用程序设计常用的功能模块的使用。

本书为每个编程主题都精心设计了完整的实例程序 (实例程序源代码可登录异步社区 <https://www.epubit.com>, 在本书页面免费下载), 通过实例程序介绍类的主要接口函数的功能和使用, 而不是简单地通过一些代码片段来孤立地解释类的使用。通过阅读本书, 练习书中的实例, 读者可以学习 Qt C++ 类库常用模块的使用方法, 学会用 Qt C++ 设计一般应用程序的方法。





# 目 录

第1章 认识 Qt	1	2.3.2 界面创建	34
1.1 Qt 简介	1	2.3.3 信号与槽的关联	37
1.2 Qt 的获取与安装	1	2.4 混合方式 UI 设计	37
1.2.1 Qt 的许可类型	1	2.4.1 设计目的	37
1.2.2 Qt 的版本	2	2.4.2 创建项目并添加资源文件	38
1.2.3 Qt 的下载与安装	2	2.4.3 设计 Action	39
1.3 Qt Creator 初步使用	5	2.4.4 设计菜单和工具栏	41
1.3.1 Qt Creator 的界面组成	5	2.4.5 代码创建其他界面组件	44
1.3.2 Qt Creator 的设置	6	2.4.6 Action 的功能实现	45
1.4 编写一个 Hello World 程序	7	2.4.7 手工创建的组件的 信号与槽	47
1.4.1 新建一个项目	7	2.4.8 为应用程序设置图标	48
1.4.2 项目的文件组成和管理	10	2.5 Qt Creator 使用技巧	48
1.4.3 项目的编译、调试与运行	11		
1.5 在 Visual Studio 里使用 Qt	13		
第2章 GUI 应用程序设计基础	16	第3章 Qt 类库概述	50
2.1 UI 文件设计与运行机制	16	3.1 Qt 核心特点	50
2.1.1 项目文件组成	16	3.1.1 概述	50
2.1.2 项目管理文件	16	3.1.2 元对象系统	50
2.1.3 界面文件	17	3.1.3 属性系统	52
2.1.4 主函数文件	19	3.1.4 信号与槽	53
2.1.5 窗体相关的文件	20	3.1.5 元对象特性测试实例	55
2.2 可视化 UI 设计	25	3.2 Qt 全局定义	59
2.2.1 实例程序功能	26	3.2.1 数据类型定义	60
2.2.2 界面组件布局	26	3.2.2 函数	60
2.2.3 信号与槽	29	3.2.3 宏定义	61
2.2.4 可视化生成槽函数原型和 框架	30	3.3 容器类	62
2.3 代码化 UI 设计	34	3.3.1 容器类概述	62
2.3.1 实例功能	34	3.3.2 顺序容器类	63
		3.3.3 关联容器类	64
		3.4 容器类的迭代	66



3.4.1	Java 类型迭代器	66	4.7	QTreeWidgetItem 和 QDockWidget	102
3.4.2	STL 类型迭代器	68	4.7.1	实例功能概述	102
3.4.3	foreach 关键字	69	4.7.2	界面设计	103
3.5	Qt 类库的模块	70	4.7.3	QTreeWidgetItem 操作	105
3.5.1	Qt 基本模块	71	4.7.4	QLabel 和 QPixmap 显示图片	110
3.5.2	Qt 附加模块	71	4.7.5	QDockWidget 的操作	111
3.5.3	增值模块	72	4.8	QTableWidget 的使用	112
3.5.4	技术预览模块	72	4.8.1	QTableWidget 概述	112
3.5.5	Qt 工具	73	4.8.2	界面设计与初始化	113
第 4 章	常用界面设计组件	74	4.8.3	QTableWidget 操作	114
4.1	字符串与输入输出	74	第 5 章	Model/View 结构	120
4.1.1	字符串与数值之间的转换	74	5.1	Model/View 结构	120
4.1.2	QString 的常用功能	76	5.1.1	Model/View 基本原理	120
4.2	SpinBox 的使用	78	5.1.2	数据模型	121
4.3	其他数值输入和显示组件	80	5.1.3	视图组件	122
4.3.1	实例功能	80	5.1.4	代理	122
4.3.2	各组件的主要功能和属性	80	5.1.5	Model/View 结构的一些 概念	123
4.3.3	实例功能的代码实现	82	5.2	QFileSystemModel	125
4.4	时间日期与定时器	84	5.2.1	QFileSystemModel 类的 基本功能	125
4.4.1	时间日期相关的类	84	5.2.2	QFileSystemModel 的 使用	125
4.4.2	日期时间数据与字符串之间的 转换	84	5.3	QStringListModel	127
4.4.3	QCalendarWidget 日历组件	87	5.3.1	QStringListModel 功能概述	127
4.4.4	定时器的使用	87	5.3.2	QStringListModel 的使用	127
4.5	QComboBox 和 QPlainTextEdit	89	5.4	QStandardItemModel	130
4.5.1	实例功能概述	89	5.4.1	功能概述	130
4.5.2	QComboBox 的使用	89	5.4.2	界面设计与主窗口类 定义	131
4.5.3	QPlainTextEdit 的使用	91	5.4.3	QStandardItemModel 的 使用	132
4.6	QListWidget 和 QToolButton	93	5.5	自定义代理	139
4.6.1	实例功能简介	93			
4.6.2	界面设计	94			
4.6.3	QListWidget 的操作	97			
4.6.4	QToolButton 与下拉式 菜单	99			
4.6.5	创建右键快捷菜单	101			





5.5.1 自定义代理的功能 .....	139	6.5 Splash 与登录窗口 .....	176
5.5.2 自定义代理类的基本 设计要求 .....	139	6.5.1 实例功能概述 .....	176
5.5.3 基于 QSpinBox 的自定义 代理类 .....	139	6.5.2 对话框界面设计和类 定义 .....	177
5.5.4 自定义代理类的使用 .....	141	6.5.3 QDlgLogin 类功能实现 .....	178
<b>第 6 章 对话框与多窗体设计</b> .....	<b>143</b>	6.5.4 Splash 登录窗口的使用 .....	181
6.1 标准对话框 .....	143	<b>第 7 章 文件系统和文件读写</b> .....	<b>182</b>
6.1.1 概述 .....	143	7.1 文本文件读写 .....	182
6.1.2 QFileDialog 对话框 .....	144	7.1.1 实例功能概述 .....	182
6.1.3 QColorDialog 对话框 .....	146	7.1.2 QFile 读写文本文件 .....	182
6.1.4 QFontDialog 对话框 .....	146	7.1.3 QFile 和 QTextStream 结合 读写文本文件 .....	184
6.1.5 QInputDialog 标准输入 对话框 .....	147	7.1.4 解决中文乱码的问题 .....	185
6.1.6 QMessageBox 消息对话框 .....	149	7.2 二进制文件读写 .....	186
6.2 自定义对话框及其调用 .....	150	7.2.1 实例功能概述 .....	186
6.2.1 对话框的不同调用方式 .....	150	7.2.2 Qt 预定义编码文件的读写 .....	187
6.2.2 对话框 QWDialogSize 的 创建和使用 .....	152	7.2.3 标准编码文件的读写 .....	192
6.2.3 对话框 QWDialogHeaders 的 创建和使用 .....	154	7.3 文件目录操作 .....	197
6.2.4 对话框 QWDialogLocate 的 创建与使用 .....	156	7.3.1 文件目录操作相关的类 .....	197
6.2.5 利用信号与槽实现交互 操作 .....	160	7.3.2 实例概述 .....	197
6.3 多窗体应用程序设计 .....	162	7.3.3 QApplication 类 .....	199
6.3.1 主要的窗体类及其用途 .....	162	7.3.4 QFile 类 .....	199
6.3.2 窗体类重要特性的设置 .....	163	7.3.5 QFileInfo 类 .....	200
6.3.3 多窗口应用程序的设计 .....	165	7.3.6 QDir 类 .....	201
6.4 MDI 应用程序设计 .....	170	7.3.7 QTemporaryDir 和 QTemporaryFile .....	203
6.4.1 MDI 简介 .....	170	7.3.8 QFileSystemWatcher 类 .....	203
6.4.2 文档窗口类 QFormDoc 的 设计 .....	171	<b>第 8 章 绘图</b> .....	<b>206</b>
6.4.3 MDI 主窗口设计与子窗口的 使用 .....	173	8.1 QPainter 基本绘图 .....	206
		8.1.1 QPainter 绘图系统 .....	206
		8.1.2 QPen 的主要功能 .....	209
		8.1.3 QBrush 的主要功能 .....	210
		8.1.4 渐变填充 .....	212
		8.1.5 QPainter 绘制基本图形 元件 .....	214



8.2 坐标系统和坐标变换 .....	217	9.3.6 百分比柱状图 .....	274
8.2.1 坐标变换函数 .....	217	9.3.7 散点图和光滑曲线图 .....	276
8.2.2 坐标变换绘图实例 .....	218	9.4 图表的其他操作 .....	277
8.2.3 视口和窗口 .....	221	9.4.1 实例功能概述 .....	277
8.2.4 绘图叠加的效果 .....	223	9.4.2 自定义 QWChartView 类 .....	278
8.3 Graphics View 绘图架构 .....	224	9.4.3 主窗口类的设计 .....	280
8.3.1 场景、视图与图形项 .....	224	9.4.4 实时显示光标处的数值 .....	281
8.3.2 Graphics View 的坐标 系统 .....	226	9.4.5 QLegendMarker 的使用 .....	282
8.3.3 Graphics View 相关的类 .....	227	9.4.6 图表的缩放 .....	283
8.3.4 Graphics View 程序基本结构 和功能实现 .....	229	<b>第 10 章 Data Visualization</b> .....	284
8.3.5 Graphics View 绘图程序 实例 .....	235	10.1 Data Visualization 模块概述 .....	284
<b>第 9 章 Qt Charts</b> .....	247	10.2 三维柱状图 .....	285
9.1 Qt Charts 概述 .....	247	10.2.1 实例功能 .....	285
9.1.1 Qt Charts 模块 .....	247	10.2.2 主窗口设计 .....	286
9.1.2 一个简单的 QChart 绘图 程序 .....	248	10.2.3 三维柱状图的创建 .....	287
9.1.3 图表的主要组成部分 .....	249	10.2.4 三维柱状图属性设置 .....	289
9.2 QChart 绘制折线图 .....	253	10.3 三维散点图 .....	293
9.2.1 实例功能 .....	253	10.3.1 绘制三维散点图 .....	293
9.2.2 主窗口类定义和初始化 .....	253	10.3.2 三维坐标轴的方向 .....	296
9.2.3 画笔设置对话框 QWDialogPen .....	256	10.3.3 散点形状与大小 .....	296
9.2.4 QChart 的设置 .....	257	10.4 三维曲面绘图 .....	296
9.2.5 QLineSeries 序列的设置 .....	259	10.4.1 三维曲面图 .....	296
9.2.6 QValueAxis 坐标轴的 设置 .....	261	10.4.2 三维地形图 .....	301
9.3 各种常见图表的绘制 .....	263	<b>第 11 章 数据库</b> .....	305
9.3.1 实例功能概述 .....	263	11.1 Qt SQL 模块概述 .....	305
9.3.2 数据准备 .....	264	11.1.1 Qt SQL 支持的数据库 .....	305
9.3.3 柱状图 .....	267	11.1.2 SQLite 数据库 .....	306
9.3.4 饼图 .....	270	11.1.3 Qt SQL 模块的主要类 .....	308
9.3.5 堆叠柱状图 .....	273	11.2 QSqlTableModel 的使用 .....	309
		11.2.1 实例功能 .....	309
		11.2.2 主窗口设计 .....	310
		11.2.3 打开数据表 .....	311
		11.2.4 添加、插入与删除记录 .....	319
		11.2.5 保存与取消修改 .....	319
		11.2.6 设置和清除照片 .....	320





11.2.7 数据记录的遍历 .....	321	12.3.2 静态链接库的使用 .....	354
11.2.8 记录排序 .....	322	12.4 创建和使用共享库 .....	357
11.2.9 记录过滤 .....	322	12.4.1 创建共享库 .....	357
11.3 QSqlQueryModel 的使用 .....	323	12.4.2 使用共享库 .....	358
11.3.1 QSqlQueryModel 功能 概述 .....	323	<b>第 13 章 多线程</b> .....	362
11.3.2 使用 QSqlQueryModel 实现 数据查询 .....	323	13.1 QThread 创建多线程程序 .....	362
11.4 QSqlQuery 的使用 .....	327	13.1.1 QThread 类功能简介 .....	362
11.4.1 QSqlQuery 基本用法 .....	327	13.1.2 掷骰子的线程 QDiceThread .....	363
11.4.2 QSqlQueryModel 和 QSqlQuery 联合使用 .....	328	13.1.3 掷骰子的多线程应用 程序 .....	365
11.5 QSqlRelationalTableModel 的 使用 .....	336	13.2 线程同步 .....	367
11.5.1 关系型数据表和实例 功能 .....	336	13.2.1 线程同步的概念 .....	367
11.5.2 关系型数据模型功能 实现 .....	338	13.2.2 基于互斥量的线程同步 .....	368
<b>第 12 章 自定义插件和库</b> .....	340	13.2.3 基于 QReadWriteLock 的 线程同步 .....	371
12.1 自定义 Widget 组件 .....	340	13.2.4 基于 QWaitCondition 的 线程同步 .....	373
12.1.1 自定义 Widget 子类 QmyBattery .....	340	13.2.5 基于信号量的线程同步 .....	377
12.1.2 自定义 Widget 组件的 使用 .....	343	<b>第 14 章 网络编程</b> .....	383
12.2 自定义 Qt Designer 插件 .....	344	14.1 主机信息查询 .....	383
12.2.1 创建 Qt Designer Widget 插件 项目 .....	344	14.1.1 QHostInfo 和 QNetworkInterface 类 .....	383
12.2.2 插件项目各文件的功能 实现 .....	346	14.1.2 QHostInfo 的使用 .....	384
12.2.3 插件的编译与安装 .....	349	14.1.3 QNetworkInterface 的 使用 .....	386
12.2.4 使用自定义插件 .....	350	14.2 TCP 通信 .....	388
12.2.5 使用 MSVC 编译器输出 中文的问题 .....	352	14.2.1 TCP 通信概述 .....	388
12.3 创建和使用静态链接库 .....	353	14.2.2 TCP 服务器端程序 设计 .....	390
12.3.1 创建静态链接库 .....	353	14.2.3 TCP 客户端程序设计 .....	395
		14.3 QUdpSocket 实现 UDP 通信 .....	397
		14.3.1 UDP 通信概述 .....	397
		14.3.2 UDP 单播和广播 .....	398
		14.3.3 UDP 组播 .....	402



14.4 基于 HTTP 协议的网络应用 程序 .....	405	15.5.3 QCamera 对象创建与 控制 .....	436
14.4.1 实现高层网络操作的类 .....	405	15.5.4 QCameraImageCapture 抓取 静态图片 .....	438
14.4.2 基于 HTTP 协议的网络 文件下载 .....	406	15.5.5 QMediaRecorder 视频 录制 .....	439
<b>第 15 章 多媒体</b> .....	409	<b>第 16 章 应用程序设计辅助功能</b> .....	441
15.1 Qt 多媒体模块功能概述 .....	409	16.1 多语言界面 .....	441
15.2 音频播放 .....	410	16.1.1 多语言界面设计概述 .....	441
15.2.1 使用 QMediaPlayer 播放 音乐文件 .....	410	16.1.2 tr() 函数的使用 .....	441
15.2.2 使用 QSoundEffect 和 Qsound 播放音效文件 .....	415	16.1.3 生成语言翻译文件 .....	442
15.3 音频输入 .....	415	16.1.4 使用 Qt Linguist 翻译 ts 文件 .....	443
15.3.1 使用 QAudioRecorder 录制音频 .....	415	16.1.5 调用翻译文件改变界面 语言 .....	444
15.3.2 使用 QAudioInput 获取 音频输入 .....	421	16.2 使用样式表自定义界面 .....	446
15.4 视频播放 .....	428	16.2.1 Qt 样式表 .....	446
15.4.1 在 QVideoWidget 上播放 视频 .....	428	16.2.2 Qt 样式表句法 .....	447
15.4.2 在 QGraphicsVideoItem 上 播放视频 .....	431	16.2.3 样式表的使用 .....	453
15.5 摄像头的使用 .....	433	16.3 使用 QStyle 设置界面外观 .....	455
15.5.1 摄像头控制概述 .....	433	16.3.1 QStyle 的作用 .....	455
15.5.2 实例主窗口设计与 初始化 .....	435	16.3.2 Qt 内置样式的使用 .....	456
		16.4 Qt 应用程序的发布 .....	457
		16.4.1 应用程序发布方式 .....	457
		16.4.2 Windows 平台上的应用 程序发布 .....	458

# 认识 Qt

## 1.1 Qt 简介

C++是一种通用的标准编程语言，使用任何编辑器都可以编写 C++源程序，然后利用 C++编译器对程序进行编译，就可以生成可执行的程序。

为了方便进行 C++程序的编写和编译，有各种综合开发环境（Integrated Developing Environment, IDE），如 Visual Studio 就是 Windows 平台上常见的编写 C++程序的 IDE。一个 IDE 不仅提供程序的编辑和编译，一般还提供一套基本类库，用于提供支持平台应用程序开发的各种基本类，如 Visual Studio 使用 MFC 进行 Windows 平台的应用程序开发。

Qt 是一套应用程序开发类库，但与 MFC 不同，Qt 是跨平台的开发类库。Qt 支持 PC 和服务器的平台，包括 Windows、Linux、macOS 等，还支持移动和嵌入式操作系统，如 iOS、Embedded Linux、Android、WinRT 等。跨平台意味着只需编写一次程序，在不同平台上无需改动或只需少许改动后再编译，就可以形成在不同平台上运行的版本。这种跨平台功能为开发者提供了极大的便利。

Qt 最早是由挪威的 Haavard Nord 和 Eirik Chambe-Eng 在 1991 年开始开发的，在 1994 年发布，并成立了一家名为 Trolltech 的公司。Trolltech 公司在 2008 年被诺基亚公司收购。2012 年，Qt 被 Digia 公司收购，并在 2014 年成立了独立的 Qt 公司，专门进行 Qt 的开发、维护和商业推广。

经过 20 多年的发展，Qt 已经成为最优秀的跨平台开发框架之一，在各行各业的项目开发中得到广泛应用。许多大型软件都是用 Qt 开发的，如 Autodesk Maya、Google Earth、Skype、WPS Office 等。

C++语言使用广泛，长盛不衰，易在不同平台上移植，其编译生成的程序执行效率高，所以在专业研究领域很多开源的算法程序或类库都是用 C++编写的。使用 Qt C++编写应用程序，可以使自己的应用程序具有跨平台移植的功能，也可以利用各种开源的类库资源。所以，扎实地掌握 Qt C++编程就如同掌握了一件利器，无论是通过编程实现自己的专业研究成果，还是从事专业软件开发都具有长远意义。

## 1.2 Qt 的获取与安装

### 1.2.1 Qt 的许可类型

Qt 的许可类型分为商业许可和开源许可，开源许可又分为 LGPLV3 和 GPLv2/GPLV3。商业许可允许开发者不公开项目的源代码，其 Qt 版本包含更多的模块（某些模块只有商业许可的版本



里才有),并能获得 Qt 公司的技术支持。当然,购买 Qt 商业许可需要支付费用。

使用开源许可的 Qt 无需支付费用,但是要遵循开源许可协议 LGPLV3 或 GPLv2/GPLV3 的规定。关于商业许可、开源许可的具体差别,开源许可的要求可以查看 Qt 官网的相关介绍。

对于 Qt 的学习来说,初学 Qt 使用开源版本的软件即可。若需要开发大型软件,并且不希望按照开源许可协议的要求公开源代码,以便对编写软件进行版权保护,则可以购买 Qt 的商业许可。

不同许可协议下,Qt 的使用权利和要求、包含的模块、工具的对比可查看 Qt 官网网址。

## 1.2.2 Qt 的版本

Qt 的版本更新比较快,且版本更新时会新增一些类或停止维护一些以前版本的类,例如 Qt 5 与 Qt 4 就有较大的区别,如果不是为了维护用旧版本编写的程序,一定要选用最新版本的 Qt 进行程序开发。

Qt 公司在 2017 年 5 月底发布了 Qt 5.9.0。Qt 5.9 是一个长期支持(long term supported, LTS)版本,在未来至少 3 年内提供更新支持,而上一个 LTS 版本是 Qt 5.6 LTS。

Qt 5.9 具有更强的性能,更好的稳定性,从 Qt 5.6 到 Qt 5.9 增加了许多新的特性,一些重要的更新如下。

- Qt 5.7 增加了 Qt 3D、Qt Quick Controls 两个模块。
- 从 Qt 5.7 开始,Qt Charts、Qt Data Visualization、Qt Virtual Keyboard、Qt Purchasing、Qt Quick 2D Renderer 等原来只在商业许可版本中存在的模块在开源许可版本中也可以使用了。
- Qt 5.8 增加了 Qt Wayland Compositor、Qt SCXML 和 Qt Serial Bus 3 个模块。
- Qt 5.9 增加了 Qt Gamepad 模块,用于不同平台上对游戏手柄的支持。
- Qt 5.9 包含一些技术预览模块,包括 Qt Remote Objects、Qt Network Authentication 和 Qt Speech。
- Qt 5.9 的 QtCore 模块增加了 qfloat16 数据类型定义。

可访问 Qt 官网页面了解 Qt 5.0 至 Qt 5.9 版本更新的历程和每个版本的新增特性描述。

由于 Qt 5.9 LTS 是一个长期技术支持版本,在未来几年里都将有更新支持,因此,本书以 Qt 5.9 LTS 版本为例进行讲解,并且所有实例程序均使用 Qt 5.9 编译测试通过。

## 1.2.3 Qt 的下载与安装

从 Qt 官网可以下载最新版本的 Qt 软件。根据开发项目的不同,Qt 分为桌面和移动设备应用开发、嵌入式设备开发两大类不同的安装包。

桌面和移动设备应用开发就是开发在 PC、服务器、手机、平板电脑等设备上运行的程序,操作系统平台可以是 Windows、Linux、macOS、Android 等。用于桌面和移动设备应用开发的 Qt 具有开源许可协议,可以免费下载和使用。

嵌入式设备开发是针对具体的嵌入式设备来开发应用程序,如物联网设备、汽车电子设备、医疗设备等特定的嵌入式设备。用于嵌入式设备开发的 Qt 可下载 30 天试用版本。

本书是介绍桌面应用程序开发的,所以下载使用的是桌面和移动设备开发的 Qt 5.9.1 开源版

本。根据 Qt 官网的提示,注册用户后才可以下载 Qt 安装程序。

Qt 5.9.1 的安装包分为在线安装包和离线安装包,为便于重复安装,最好下载离线安装包。离线安装包根据使用的操作系统平台不同,分为 Linux、macOS 和 Windows 3 个版本,本书实例都是用 Windows 7 平台上的 Qt 开发的,所以这里下载 Windows 版本的 Qt 5.9.1 离线安装包。

Qt 5.9 以前版本的离线安装包即使是在 Windows 平台上,也会根据使用的编译器不同分为很多版本,如 MinGW 32-bit 版本、MSVC2015 32-bit 版本、MSVC2015 64-bit 版本等。而 Qt 5.9 在一个平台上只有一个安装包,编译器的选择放在了安装过程里,所以下载的 Windows 平台上的 Qt 5.9.1 安装包只有一个可执行文件。

双击下载后的 Qt 5.9.1 离线安装包可执行文件,就开始执行安装过程,安装过程与一般的 Windows 应用程序一样,按照向导进行操作即可。

在安装过程中会出现如图 1-1 所示的安装选项设置页面,在这个页面里选择需要安装的模块。“Qt 5.9.1”节点下面是 Qt 的功能模块,包括用于不同编译器和平台的模块,这些模块包括内容如下。

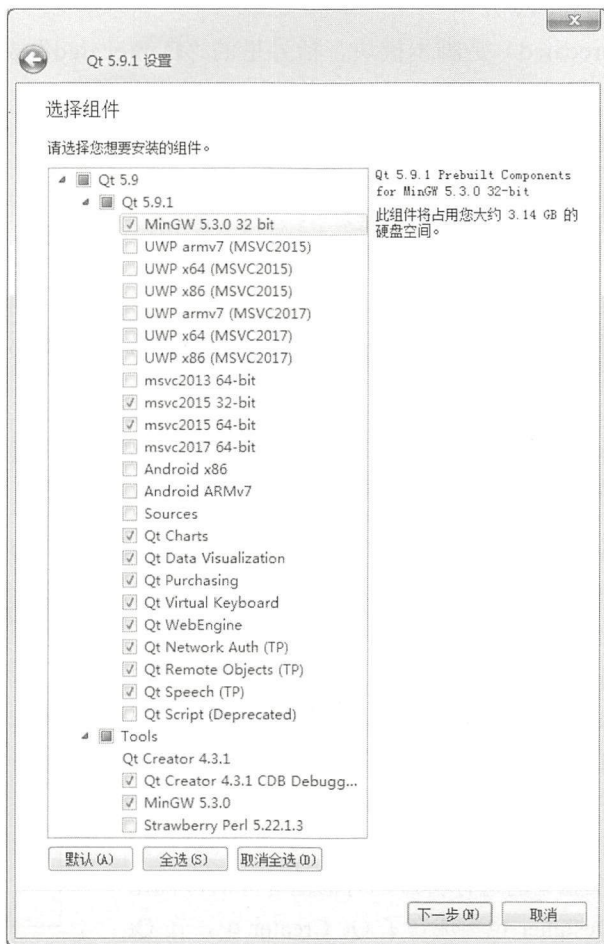


图 1-1 Qt 5.9.1 安装选项设置页面

- MinGW 5.3.0 32 bit 编译器模块。MinGW 是 Minimalist GNU for Windows 的缩写，MinGW 是 Windows 平台上使用的 GNU 工具集导入库的集合。
- 用于 UWP 编译的模块。UWP 是 Windows 10 中 Universal Windows Platform 的简称，有不同编译器类型的 UWP。
- 用于 windows 平台上的 MSVC 编译器模块，如 msvc2015 32-bit 和 msvc2015 64-bit 等。要安装 MSVC 编译器的模块，需要计算机上已经安装相应版本的 Visual Studio。
- 用于 Android 平台的模块，如 Android x86 和 Android ARMv7。
- Sources 是 Qt 的源程序。
- Qt Charts 是二维图表模块，用于绘制柱状图、饼图、曲线图等常用二维图表。
- Qt Data Visualization 是三维数据图表模块，用于数据的三维显示，如散点的三维空间分布、三维曲面等。
- Qt Purchasing、Qt WebEngine、Qt Network Auth(TP)等其他模块，括号里的 TP 表示技术预览 (Technology Preview)。
- Qt Scrip (Deprecated) 是脚本模块，括号里的 “Deprecated” 表示这是个已经过时的模块。

“Tools” 节点下面是一些工具软件，包括内容如下。

- Qt Creator 4.3.1 是用于 Qt 程序开发的 IDE。
- MinGW 5.3.0 是 MinGW 编译工具链。
- Strawberry Perl 是一个 Perl 语言工具。

根据个人的需要设置安装选项，无需选择所有的安装选项。例如，如果不需要进行 UWP 平台的开发，UWP 模块就可以都不选；如果不是为和以前开发的源程序兼容，过时的模块不要选择，如 Qt Script 就是已过时的模块。

---

**注意** 如果选择安装 MSVC 编译器的模块，需要在计算机上安装相应的 Microsoft Visual Studio 开发工具，使用免费的 Community 版本的 Visual Studio 即可。

---

安装完成后，在 Windows “开始” 菜单里建立的 Qt 5.9.1 程序组内容如图 1-2 所示。程序组中一个主要的程序是 Qt Creator 4.3.1(Community)，它是用于开发 Qt 程序的 IDE，是 Qt 的主要工具软件。

根据选择安装的编译器模块会建立几个子分组，见图 1-2 中的 MinGW 5.3.0 (32-bit)、MSVC 2015 (32-bit) 和 MSVC 2015 (64-bit)，每个分组下面主要有 3 个工具软件。

- Assistant 是一个独立的查看 Qt 帮助文件的程序，集成在了 Qt Creator 中。
- Designer 是一个独立的进行窗口、对话框等界面可视化设计的程序。Designer 也集成在了 Qt Creator 中，在 Qt Creator 中编辑或创建界面文件时，就可以自动打开并进

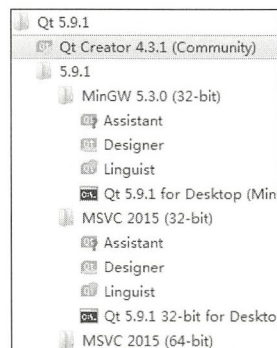


图 1-2 安装后“开始”菜单里的 Qt 5.9.1 程序组



行界面设计。

- Linguist 是一个编辑语言资源文件的程序，在开发多语言界面的应用程序时会用到。

这 3 个工具软件可独立使用，前两个集成到了 Qt Creator 里，可在 Qt Creator 打开。所以 Qt 的主要工具是 Qt Creator，要编写 Qt 程序，运行 Qt Creator 即可。

## 1.3 Qt Creator 初步使用

### 1.3.1 Qt Creator 的界面组成

启动 Qt Creator，出现如图 1-3 所示的主窗口。

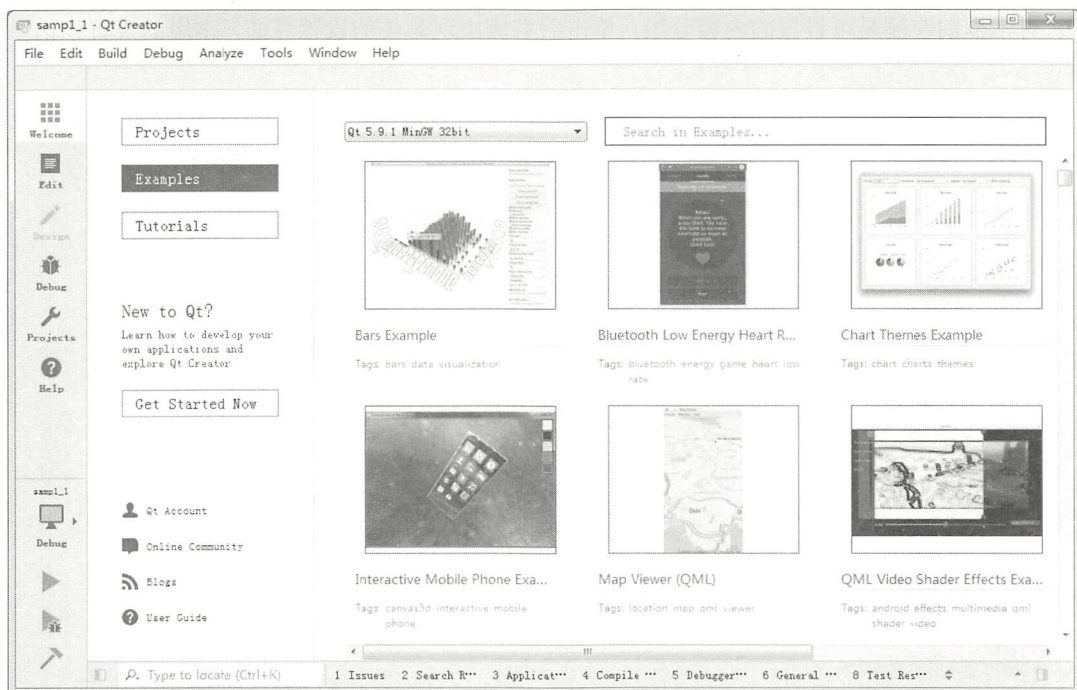


图 1-3 Qt Creator 主窗口

Qt Creator 的界面很简洁。上方是主菜单栏，左侧是主工具栏，窗口的中间部分是工作区。根据设计内容不同，工作区会显示不同的内容。

图 1-3 是在左侧主工具栏单击“Welcome”按钮后显示实例的界面。这时工作区的左侧有“Projects”“Examples”“Tutorials”“Get Started Now”几个按钮，单击后会在主工作区显示相应的内容。

- 单击“Projects”按钮后，工作区显示新建项目按钮和最近打开项目的列表。
- 单击“Examples”按钮后，工作区显示 Qt 自带的大量实例，选择某个实例就可以在 Qt Creator 中打开该项目源程序。

- 单击“Tutorials”按钮后，工作区显示各种视频教程，查看视频教程需要联网并使用浏览器打开。
- 单击“Get Started Now”按钮，工作区显示“Qt Creator Manual”帮助主题内容。

主窗口左侧是主工具栏，主工具栏提供了项目文件编辑、窗体设计、程序调试、项目设置等各种功能按钮。

### 1.3.2 Qt Creator 的设置

对 Qt Creator 可以进行一些设置，如刚安装好的 Qt Creator 界面语言可能是中文，但是很多词汇翻译得并不恰当，可以将 Qt Creator 的界面语言设置为英文。

单击 Qt Creator 菜单栏的“Tools”→“Options”菜单项会打开选项设置对话框（如图 1-4 所示）。对话框的左侧是可设置的内容分组，单击后右侧出现具体的设置界面。常用的设置包括以下几点。

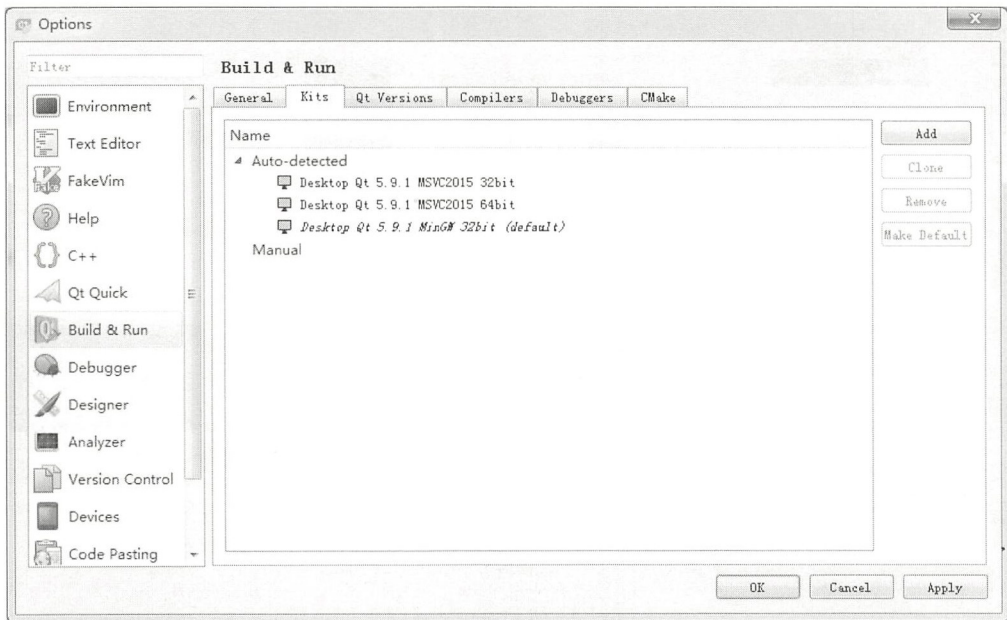


图 1-4 Options 的 Build & Run 设置页面

1. Environment 设置：在 Interface 页面可以设置语言和主题，本书全部以英文界面的 Qt Creator 进行讲解，所以语言选择为 English；为了使界面抓图更清晰，设置主题为 Flat Light。更改语言和主题后需要重新启动 Qt Creator 才会生效。

2. Text Editor 设置：在此界面可以设置文本编辑器的字体，设置各种类型文字的字体颜色，如关键字、数字、字符串、注释等字体颜色，也可以选择不同的配色主题。编辑器缺省字体的大小为 9，可以修改得大一些。

3. Build & Run 设置：图 1-4 显示的是 Build & Run 的设置界面，它有以下几个页面。

- (1) Kits 页面显示 Qt Creator 可用的编译工具，在图中可以看到有 3 个编译工具可用。
- (2) Qt Versions 页面显示安装的 Qt 版本，有 Qt 5.9.1 MinGW 32bit、Qt 5.9.1 MSVC2015 32bit 和 Qt 5.9.1 MSVC2015 64bit 3 个可用的版本。
- (3) Compilers 页面显示系统里可用的 C 和 C++ 编译器，由于安装了 MinGW 和 Visual Studio 2015，Qt Creator 会自动检测出这些编译器。
- (4) Debuggers 页面显示 Qt Creator 自动检测到的调试器，有 GNU gdb for MinGW 调试器和 Windows 的 CDB 调试器。

**注意** 如果只是在计算机上安装了 Visual Studio 2015，图 1-4 显示的界面上 MSVC2015 的两个编译器的图标会变为带有感叹号的一个黄色图标。Debuggers 页面没有 Windows 的 CDB 调试器，可以用 MSVC 编译器对 Qt Creator 编写的程序进行编译，但是不能调试，这是因为缺少了 Windows Software Development Kit (SDK)。这个 SDK 不会随 Visual Studio 一同安装，需要从 Microsoft 网站上下载。可以下载 Windows Software Development Kit (SDK) for Windows 8.1，安装后重启计算机即可。

## 1.4 编写一个 Hello World 程序

学习一种编程语言或编程环境，通常会先编写一个“Hello World”程序。我们也用 Qt Creator 编写一个“Hello World”程序，以初步了解 Qt Creator 设计应用程序的基本过程，对使用 Qt Creator 编写 Qt C++ 应用程序建立初步的了解。

### 1.4.1 新建一个项目

单击 Qt Creator 的菜单项“File”→“New File or Project”，出现如图 1-5 所示的对话框。在这个对话框里选择需要创建的项目或文件的模板。

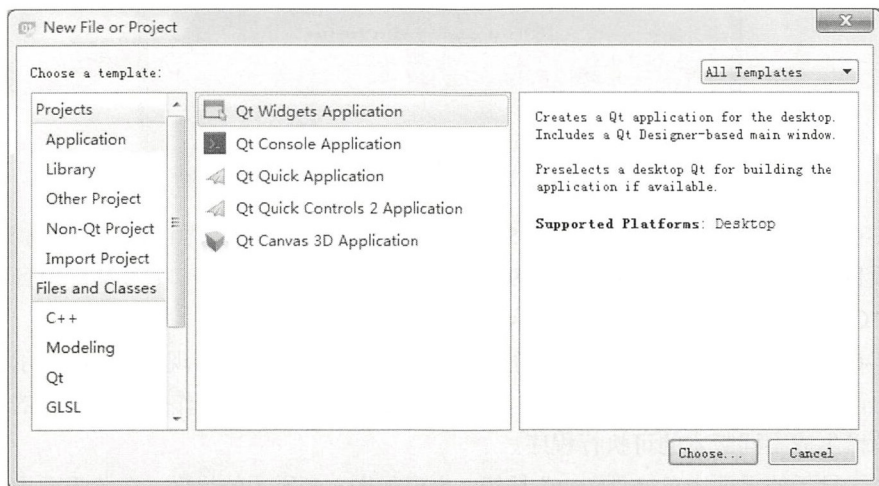


图 1-5 新建文件或项目对话框



Qt Creator 可以创建多种项目，在最左侧的列表框中单击“Application”，中间的列表框中列出了可以创建的应用程序的模板，各类应用程序如下。

- Qt Widgets Application，支持桌面平台的有图形用户界面（Graphic User Interface，GUI）界面的应用程序。GUI 的设计完全基于 C++ 语言，采用 Qt 提供的一套 C++ 类库。
- Qt Console Application，控制台应用程序，无 GUI 界面，一般用于学习 C/C++ 语言，只需要简单的输入输出操作时可创建此类项目。
- Qt Quick Application，创建可部署的 Qt Quick 2 应用程序。Qt Quick 是 Qt 支持的一套 GUI 开发架构，其界面设计采用 QML 语言，程序架构采用 C++ 语言。利用 Qt Quick 可以设计非常炫的用户界面，一般用于移动设备或嵌入式设备上无边框的应用程序的设计。
- Qt Quick Controls 2 Application，创建基于 Qt Quick Controls 2 组件的可部署的 Qt Quick 2 应用程序。Qt Quick Controls 2 组件只有 Qt 5.7 及以后版本才有。
- Qt Canvas 3D Application，创建 Qt Canvas 3D QML 项目，也是基于 QML 语言的界面设计，支持 3D 画布。

在图 1-5 显示的对话框中选择项目类型为 Qt Widgets Application 后，单击“Choose...”按钮，出现如图 1-6 所示的新建项目向导。

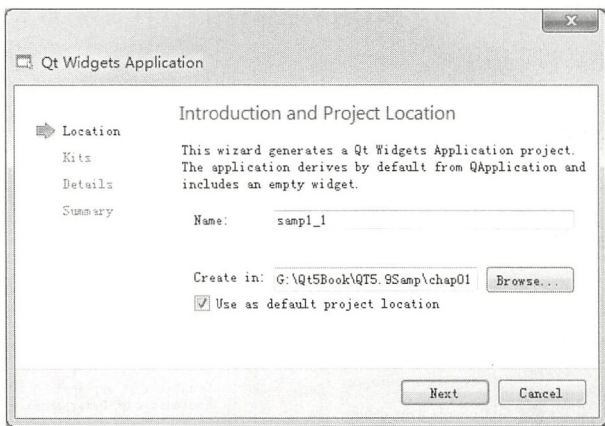


图 1-6 新建项目向导第 1 步：项目名称和存储路径设置

在图 1-6 中，选择一个目录，如“G:\Qt5Book\Qt5.9Samp\chap01”，再设置项目名称为 samp1\_1，这样新建项目后，会在“G:\Qt5Book\Qt5.9Samp\chap01”目录下新建一个目录，项目所有文件保存在目录“G:\Qt5Book\Qt5.9Samp\chap01\samp1\_1\”下。

在图 1-6 中设置好项目名称和保存路径后，单击“Next”按钮，出现如图 1-7 所示的选择编译工具的界面，可以将 3 个编译工具都选中，在编译项目时再选择一个作为当前使用的编译工具，这样可以编译生成不同版本的可执行程序。

在图 1-7 显示的界面中单击“Next”按钮，出现如图 1-8 所示的界面。在此界面中选择需要创建界面的基类（base class）。有 3 种基类可以选择：

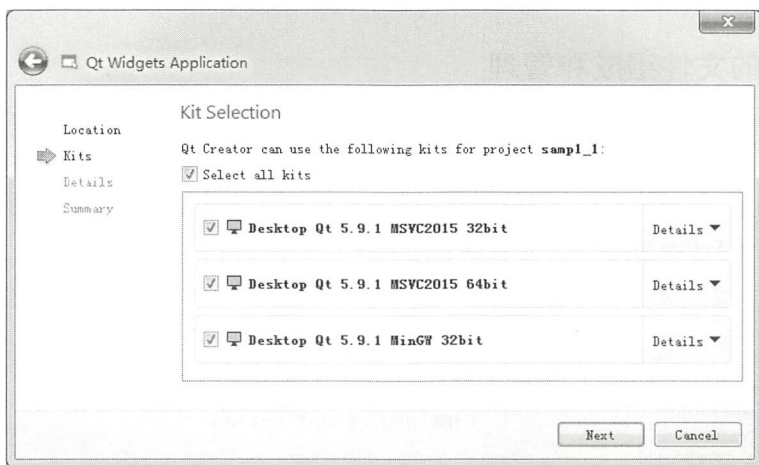


图 1-7 新建项目向导第 2 步：选择编译工具

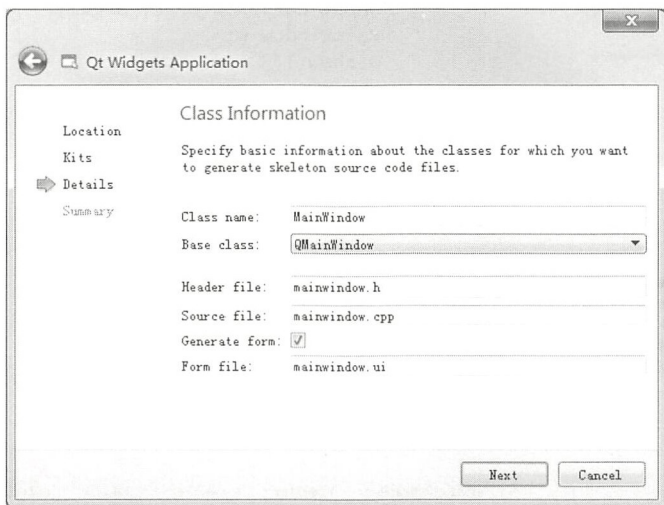


图 1-8 新建项目向导第 3 步：选择界面基类

- QMainWindow 是主窗口类，主窗口具有主菜单栏、工具栏和状态栏，类似于一般的应用程序的主窗口；
- QWidget 是所有具有可视界面类的基类，选择 QWidget 创建的界面对各种界面组件都可以支持；
- QDialog 是对话框类，可建立一个基于对话框的界面。

在此选择 QMainWindow 作为基类，自动更改的各个文件名不用手动去修改。勾选“Generate form”复选框。这个选项如果勾选，就会由 Qt Creator 创建用户界面（User Interface，UI）文件，否则，需要自己编程手工创建界面。初始学习，为了了解 Qt Creator 的设计功能，勾选此选项。然后单击“Next”按钮，出现一个页面，总结了需要创建的文件和文件保存目录，单击“Finish”按钮就可以完成项目的创建。

## 1.4.2 项目的文件组成和管理

完成了以上新建项目的步骤后,在 Qt Creator 的左侧工具栏中单击“Edit”按钮,可显示如图 1-9 所示的窗口。窗口左侧有上下两个子窗口,上方的目录树显示了项目内文件的组织结构,显示当前项目为 samp1\_1。项目的名称构成目录树的一个根节点,Qt Creator 可以打开多个项目,但是只有一个活动项目(Active Project),活动项目的项目名称节点用粗体字体表示。

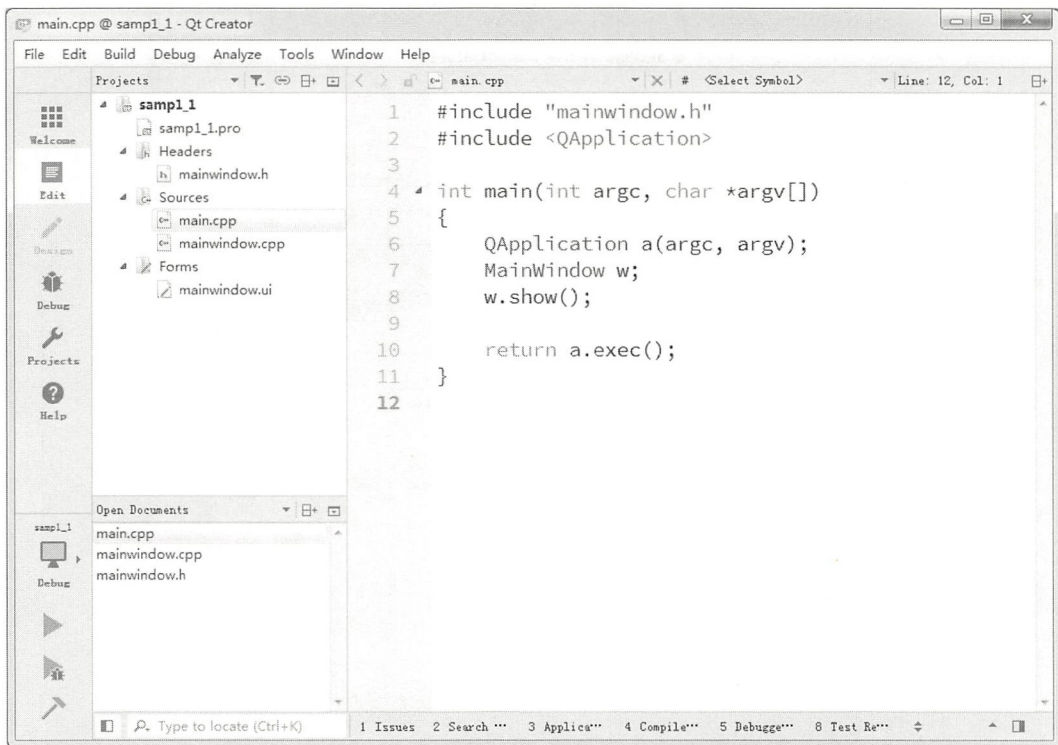


图 1-9 项目管理与文件编辑界面

在项目名称节点下面,分组管理着项目内的各种源文件,几个文件及分组分别为以下几项。

- samp1\_1.pro 是项目管理文件,包括一些对项目的设置项。
- Headers 分组,该节点下是项目内的所有头文件(.h),图 1-9 中所示项目有一个头文件 mainwindow.h,是主窗口类的头文件。
- Sources 分组:该节点下是项目内的所有 C++源文件(.cpp),图 1-9 中所示项目有两个 C++源文件,mainwindow.cpp 是主窗口类的实现文件,与 mainwindow.h 文件对应。main.cpp 是主函数文件,也是应用程序的入口。
- Forms 分组:该节点下是项目内的所有界面文件(.ui)。图 1-9 中所示项目有一个界面文件 mainwindow.ui,是主窗口的界面文件。界面文件是文本文件,使用 XML 语言描述界面的组成。



左侧上下两个子窗口的显示内容可以通过其上方的一个下拉列表框进行选择，可以选择的显示内容包括 Projects、Open Documents、Bookmarks、File System、Class View、Outline 等。在图 1-9 中，上方的子窗口显示了项目的文件目录树，下方显示打开的文件列表。可以在下方选择显示 Class View，这样下方则显示项目内所有的类的结构，便于程序浏览和快速切换到需要的代码位置。

双击文件目录树中的文件/mainwindow.ui，出现如图 1-10 所示的窗体设计界面，这个界面实际上是 Qt Creator 中集成的 Qt Designer。窗口左侧是分组的组件面板，中间是设计的窗体。在组件面板的 Display Widgets 分组里，将一个 Label 组件拖放到设计的窗体上面。双击刚刚放置的 Label 组件，可以编辑其文字内容，将文字内容更改为“Hello, World!”。还可以在窗口右下方的属性编辑器（Property Editor）里编辑标签的 Font 属性，Point Size 更改为 12，勾选 Bold。

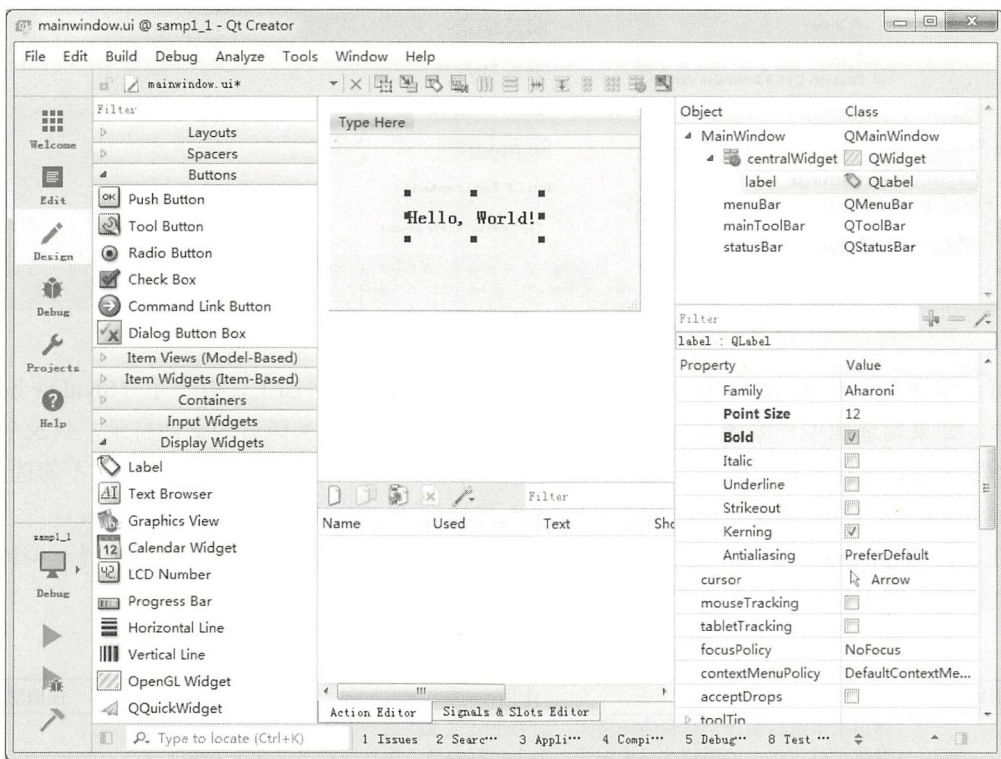


图 1-10 集成在 Qt Creator 中 UI 设计器

### 1.4.3 项目的编译、调试与运行

单击主窗口左侧工具栏上的“Projects”按钮，出现如图 1-11 所示的项目编译设置界面。

界面左侧一栏的“Build & Run”下面显示了本项目中可用的编译器工具，要使用哪一个编译器用于项目编译，单击其名称即可，选择的编译器名称会用粗体字表示。这里选择使用 MinGW 32bit 编译器。

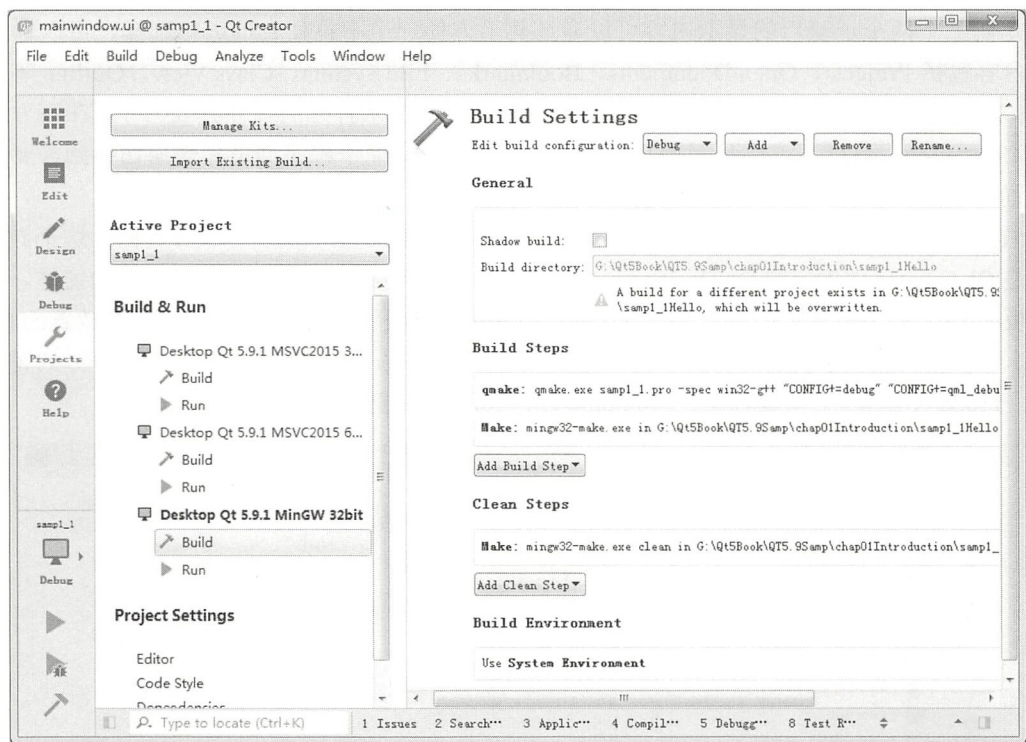


图 1-11 项目编译器选择和设置界面

每个编译器又有 Build 和 Run 两个设置界面。在 Build 设置界面上，有一个“Shadow build”复选框。如果勾选此项，编译后将在项目的同级目录下建立一个编译后的文件目录，目录名称包含编译器信息，这种方式一般用于使用不同编译器创建不同版本的可执行文件。如果不勾选此项，编译后将在项目的目录下建立“Debug”和“Release”子目录用于存放编译后的文件。

在设计完 mainwindow.ui 文件，并设置好编译工具之后，就可以对项目进行编译、调试或运行。主窗口左侧工具栏下方有 4 个按钮，其功能见表 1-1。

表 1-1 编译调试工具栏按钮的作用

图标	作用	快捷键
	弹出菜单选择编译工具和编译模式，如 Debug 或 Release 模式	
	直接运行程序，如果修改后未编译，会先进行编译。即使在程序中设置了断点，此方式运行的程序也无法调试。	Ctrl+R
	项目需要以 Debug 模式编译，点此按钮开始调试运行，可以在程序中设置断点。若是以 Release 模式编译，点此按钮也无法进行调试。	F5
	编译当前项目	Ctrl+B

首先对项目进行编译，没有错误后，再运行程序。程序运行的界面如图 1-12 所示。这就是一个标准的桌面应用程序，我们采用可视化的方式设计了一个窗口，并在上面显示了字符串“Hello, World!”。

在 Qt Creator 中也可以对程序设置断点进行调试，但是必须以 Debug 模式编译，并以“Start

Debugging”（快捷键 F5）方式运行程序。程序调试的方法与一般 IDE 工具类似，不再详述。注意，要在 Qt Creator 里调试 MSVC2015 编译的程序，必须安装 Windows 软件开发工具包 SDK。

在图 1-11 的界面中选择其他编译器，并且勾选“Shadow build”，用 Debug 和 Release 模式分别编译，将会在项目的同级目录下生成对应的目录，保存编译后的文件。图 1-13 显示的是实例 samp1\_1 采用 3 种编译器，分别用 Debug 和 Release 模式编译后生成的目录结构。3 种编译器，2 种编译模式，生成了 6 个文件夹。

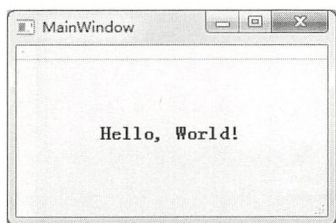


图 1-12 实例程序 samp1\_1 运行时界面



图 1-13 勾选“Shadow build”选项后使用不同编译器和编译模式生成的输出目录

**提示** 除非特别说明，本书后面的实例程序均在 Windows 7.0 SP1 64 位平台上用 Qt Creator 4.3.1 开发，采用 Qt 5.9.1 的 MinGW 32bit 编译器进行编译。只有在第 12 章设计 Qt Designer 的 Widget 插件时才必须使用 MSVC2015 32bit 编译器，在第 15.5 节使用摄像头录像时必须使用 Linux 平台的 Qt。

## 1.5 在 Visual Studio 里使用 Qt

在 Qt Creator 中可以使用 MSVC 编译工具对 Qt 项目进行编译。若有人比较习惯于使用 Visual Studio，或某些项目必须使用 Visual Studio，也可以在 Visual Studio 里创建和管理 Qt 程序项目。要在 Visual Studio 中使用 Qt，需要安装一个 Visual Studio 的 Qt 插件，这个插件程序由 Qt 公司提供。

目前最新的 Visual Studio Qt 插件是“Visual Studio Add-in 2.0.0 for Qt5 MSVC 2015”，可以从 Qt 官网下载并安装。安装此插件之前，需已经安装好 Visual Studio 2015。

这里省略 Visual Studio Add-in 2.0.0 for Qt5 的安装过程。安装完成后，在 Visual Studio 的主菜单栏上增加了一个菜单组“Qt VS Tools”，在新建项目向导里增加了可创建 Qt 项目的项目模板。

在 Visual Studio 2015 里创建一个 Qt GUI 应用程序项目。创建项目时选择项目模板的对话框如图 1-14 所示，选择创建 Qt GUI Application 项目，根据向导提示完成项目 samp1\_2 的创建。

按照向导设置创建完项目后，Visual Studio 管理项目的全部文件，有一个 samp1\_2.ui 的窗体文件，双击此文件，会自动使用 Qt Designer 打开窗体文件进行界面设计，如同在 Qt Creator 里设计窗体一样。

在首次使用 Visual Studio 编译 Qt 项目之前，必须先进行一些设置，否则会提示没有设置 Qt



版本，无法编译项目。

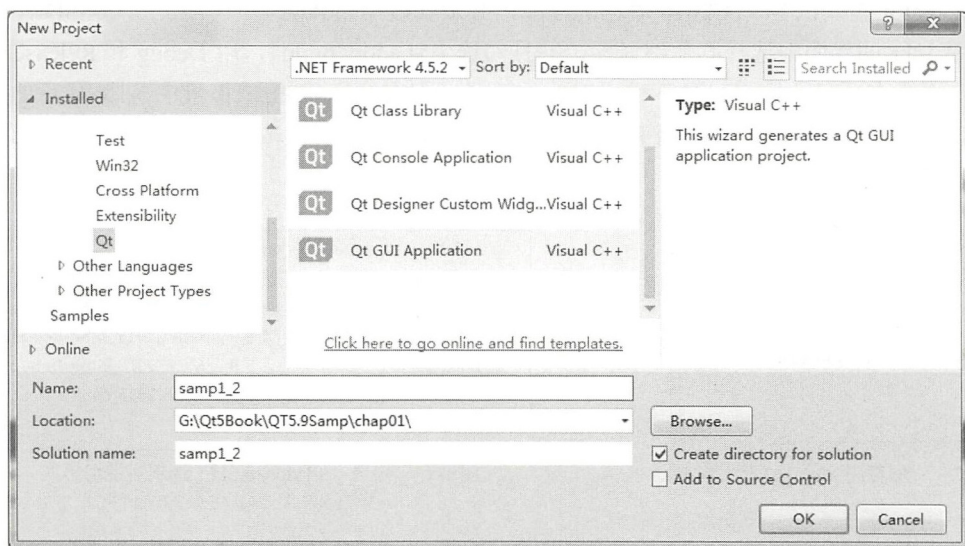


图 1-14 在 Visual Studio 2015 里创建 Qt 项目 samp1\_2

首先要设置 Qt 版本。单击 Visual Studio 菜单项“Qt VS Tools”→“Qt Options”，出现如图 1-15 所示的对话框。Qt Versions 页面显示了可以使用的 Qt 版本（这是已经设置好的界面），在未设置之前，框里是空白的。单击“Add”按钮出现如图 1-16 所示的添加 Qt 版本对话框。

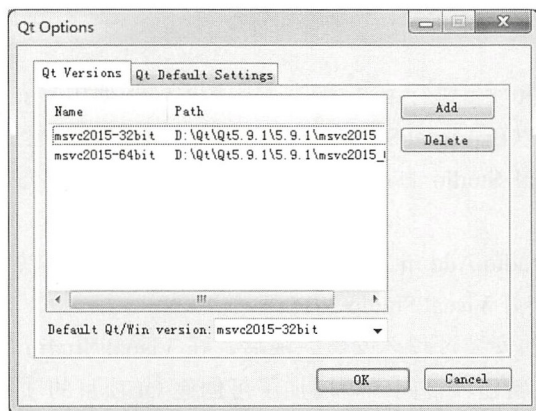


图 1-15 Qt Options 设置对话框

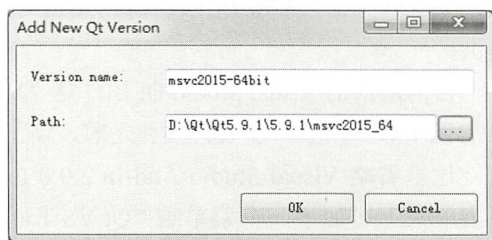


图 1-16 添加 Qt 版本对话框

单击“Path”文本框后面的按钮，在出现的目录选择对话框里选择 Qt 5.9.1 安装目录下的 MSVC 编译器目录，如“D:\Qt\Qt5.9.1\5.9.1\msvc2015\_64”。选择目录后，Version name 编辑框里会自动出现版本名称，可以修改此名称为意义更明显的名字，如“msvc2015-64bit”。

然后，再单击 Visual Studio 菜单项“Qt VS Tools”→“Qt Project Settings”，为项目设置 Qt 版本，出现如图 1-17 所示的对话框。在此对话框的 Properties 分页下的列表框里，在 Version 下拉列表框中选择某个 Qt 版本。

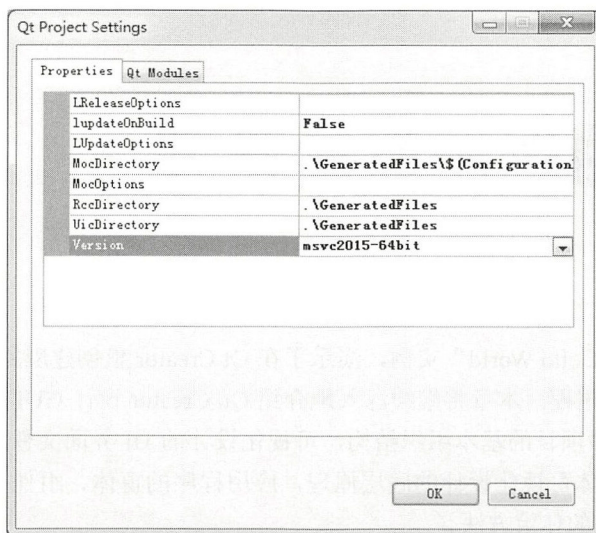


图 1-17 Qt 项目设置对话框

完成这两项设置后，再进行编译就没有问题了。项目的运行、调试等就都是 Visual Studio 的操作了，这里不再赘述。

---

**提示** 在 Qt Creator 里就可以使用 MSVC 编译器对项目进行编译，并不是只有在 Visual Studio 里才可以使用 MSVC 编译器编译 Qt 的项目。

---



## 第2章

# GUI 应用程序设计基础

上一章通过一个“Hello World”实例，演示了在 Qt Creator 里创建应用程序、设计窗体界面、编译和运行程序的基本过程。本章将继续深入地介绍 Qt Creator 设计 GUI 应用程序的基本方法，包括 Qt 创建的应用程序项目的基本组织结构，可视化设计的 UI 界面文件的原理和运行机制，信号与槽的使用方法，窗体可视化设计的底层原理，应用程序的窗体、组件布局、菜单、工具栏、Actions 等常见设计元素的使用方法。

## 2.1 UI 文件设计与运行机制

### 2.1.1 项目文件组成

在 Qt Creator 中新建一个 Widget Application 项目 samp2\_1，在选择窗口基类的页面（图 1-8）选择 QWidget 作为窗体基类，并选中“Generate form”复选框。创建后的项目文件目录树如图 2-1 所示。

这个项目包含以下一些文件。

- 项目组织文件 samp2\_1.pro，存储项目设置的文件。
- 主程序入口文件 main.cpp，实现 main()函数的程序文件。
- 窗体界面文件 widget.ui，一个 XML 格式存储的窗体上的元件及其布局的文件。
- widget.h 是所设计的窗体类的头文件，widget.cpp 是 widget.h 里定义类的实现文件。在 C++里，任何窗体或界面组件都是用类封装的，一个类一般有一个头文件（.h 文件）和一个源程序文件（.cpp 文件）。

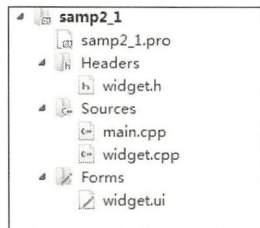


图 2-1 项目文件的目录树

### 2.1.2 项目管理文件

后缀为“.pro”的文件是项目的管理文件，文件名就是项目的名称，如本项目中的 samp2\_1.pro。下面是 samp2\_1.pro 文件的内容。

```
Qt += core gui
greaterThan(Qt_MAJOR_VERSION, 4): Qt += widgets

TARGET = samp2_1
```





```

TEMPLATE = app
SOURCES += main.cpp\
           widget.cpp
HEADERS  += widget.h
FORMS    += widget.ui

```

项目管理文件用于记录项目的一些设置，以及项目包含文件的组织管理。

“Qt += core gui”表示项目中加入 core gui 模块。core gui 是 Qt 用于 GUI 设计的类库模块，如果创建的是控制台（Console）应用程序，就不需要添加 core gui。

Qt 类库以模块的形式组织各种功能的类，根据项目涉及的功能需求，在项目中添加适当的类库模块支持。例如，如果项目中使用到了涉及数据库操作的类就需要用到 sql 模块，在 pro 文件中需要增加如下一行：

```
Qt +=sql
```

samp2\_1.pro 中的第 2 行是：

```
greaterThan(Qt_MAJOR_VERSION, 4): Qt += widgets
```

这是个条件执行语句，表示当 Qt 主版本大于 4 时，才加入 widgets 模块。

“TARGET = samp2\_1”表示生成的目标可执行文件的名称，即编译后生成的可执行文件是 samp2\_1.exe。

“TEMPLATE = app”表示项目使用的模板是 app，是一般的应用程序。

后面的 SOURCES、HEADERS、FORMS 记录了项目中包含的源程序文件、头文件和窗体文件（.ui 文件）的名称。这些文件列表是 Qt Creator 自动添加到项目管理文件里面的，用户不需要手动修改。当添加一个文件到项目，或从项目里删除一个文件时，项目管理文件里的条目会自动修改。

### 2.1.3 界面文件

后缀为“.ui”的文件是可视化设计的窗体的定义文件，如 widget.ui。双击项目文件目录树中的文件 widget.ui，会打开一个集成在 Qt Creator 中的 Qt Designer 对窗体进行可视化设计，如图 2-2 所示。

---

约定 本书后面将称这个集成在 Qt Creator 中的 Qt Designer 为“UI 设计器”，以便与独立运行的 Qt Designer 区别开来。

---

图 2-2 中的 UI 设计器有以下一些功能区域。

- 组件面板。窗口左侧是界面设计组件面板，分为多个组，如 Layouts、Buttons、Display Widgets 等，界面设计的常见组件都可以在组件面板里找到。
- 中间主要区域是待设计的窗体。如果要将某个组件放置到窗体上时，从组件面板上拖放一个组件到窗体上即可。例如，先放一个 Label 和一个 Push Button 到窗体上。
- Signals 和 Slots 编辑器与 Action 编辑器是位于待设计窗体下方的两个编辑器。Signals 和 Slots 编辑器用于可视化地进行信号与槽的关联，Action 编辑器用于可视化设计 Action。

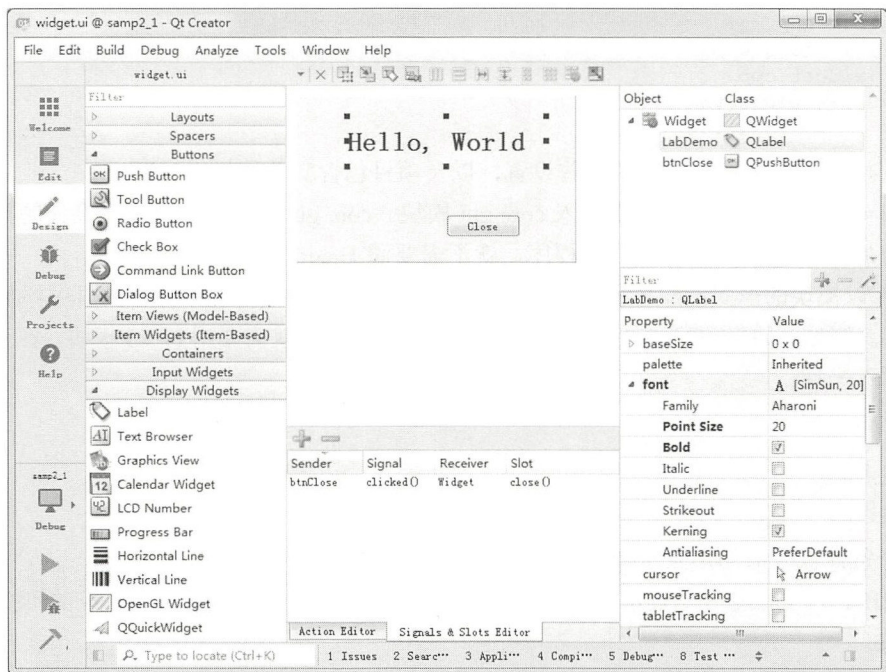


图 2-2 集成在 Qt Creator 中的 UI 设计器

- 布局 and 界面设计工具栏。窗口上方的一个工具栏，工具栏上的按钮主要实现布局 and 界面设计。
- 对象浏览器（Object Inspector）。窗口右上方是 Object Inspector，用树状视图显示窗体上各组件之间的布局包含关系，视图有两列，显示每个组件的对象名称（ObjectName）和类名称。
- 属性编辑器（Property Editor）。窗口右下方是属性编辑器，是界面设计时最常用到的编辑器。属性编辑器显示某个选中的组件或窗体的各种属性及其取值，可以在属性编辑器里修改这些属性的值。

图 2-3 显示的是选中窗体上放置的标签组件后属性编辑器的内容。最上方显示的文字“LabDemo: QLabel”表示这个组件是一个 QLabel 类的组件，objectName 是 LabDemo。属性编辑器的内容分为两列，Property 列是属性的名称，Value 列是属性的值。属性又分为多个组，实际上表示了类的继承关系，如在图 2-3 中，可以看出 QLabel 的继承关系是 QObject→QWidget→QFrame→QLabel。

objectName 表示组件的对象名称，界面上的每个组件都需要一个唯一的对象名称，以便被引用。界面上的组件的命名应该遵循一定的法则，具体使用什么样的命名法则根据个人习惯而定，主要目的是便于区分和记忆，也要便于与普通变量相区分。

设置其他属性的值只需在属性编辑器里操作即可，如设置 LabDemo 的 text 属性为“Hello, World”，只需像图 2-3 那样修改 text 属性的值即可。

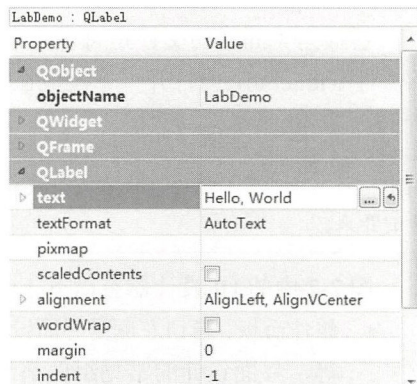


图 2-3 界面组件的属性编辑器



**提示** 标准 C++ 语言里并没有 property 关键字，property 是 Qt 对标准 C++ 的扩展，使得在 Qt Designer 里就可以可视化设置类的数据。

在图 2-2 显示的设计窗体上，放置一个 Label 和一个 Push Button 组件，它们的主要属性设置见表 2-1。

表 2-1 界面组件的属性设置

ObjectName	类名称	属性设置	备注
LabDemo	QLabel	Text=" Hello, World" Font.PointSize=20 Font.bold=true	设置标签显示文字和字体
btnClose	QPushButton	Text=" Close"	设置按钮的文字

编辑完属性之后，再为 btnClose 按钮增加一个功能，就是单击此按钮时，关闭窗口，退出程序。使用 Signals 和 Slots 编辑器完成这个功能，如图 2-4 所示。

在信号与槽编辑器的工具栏上单击“Add”按钮，在出现的条目中，Sender 选择 btnClose，Signal 选择 clicked()，Receiver 选择窗体 Widget，Slot 选择 close()。这样设置表示当按钮 btnClose 被单击时，就执行 Widget 的 close() 函数，实现关闭窗口的功能。

然后对项目进行编译和运行，可以出现如图 2-5 所示的窗口，单击“Close”按钮可以关闭程序。标签的文字内容和字体被修改了，窗口标题也显示为所设置的标题，而我们并没有编写一行程序语句，Qt 是怎么实现这些功能的呢？

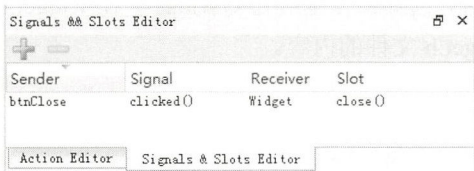


图 2-4 信号与槽编辑器中设计信号与槽的关联

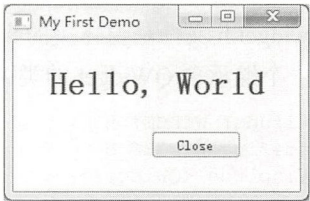


图 2-5 具有 Close 按钮的“Hello World”程序

### 2.1.4 主函数文件

main.cpp 是实现 main() 函数的文件，下面是 main.cpp 文件的内容。

```
#include "widget.h"
#include <QApplication>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv); //定义并创建应用程序
    Widget w; //定义并创建窗口
    w.show(); //显示窗口
    return a.exec(); //应用程序运行
}
```

main() 函数是应用程序的入口。它的主要功能是创建应用程序，创建窗口，显示窗口，并运行应用程序，开始应用程序的消息循环和事件处理。





QApplication 是 Qt 的标准应用程序类，第 1 行代码定义了一个 QApplication 类的实例 a，就是应用程序对象。

然后定义了一个 Widget 类的变量 w，Widget 是本实例设计的窗口的类名，定义此窗口后再用 w.show() 显示此窗口。

最后一行用 a.exec() 启动应用程序的执行，开始应用程序的消息循环和事件处理。

### 2.1.5 窗体相关的文件

为了搞清楚窗体类的定义，以及界面功能的实现原理，这里将项目进行编译。编译后在项目目录下会自动生成一个文件 ui\_widget.h，这样对于一个窗体，就有 4 个文件了，各文件的功能说明见表 2-2。

表 2-2 窗体相关的 4 个文件

文件	功能
widget.h	定义窗体类的头文件，定义了类 Widget
widget.cpp	Widget 类的功能实现源程序文件
widget.ui	窗体界面文件，由 UI 设计器自动生成，存储了窗体上各个组件的属性设置和布局
ui_widget.h	编译后，根据窗体上的组件及其属性、信号与槽的关联等自动生成的一个类的定义文件，类的名称是 Ui_Widget

下面分别分析各个文件的内容及其功能，以及它们是如何联系在一起工作，实现界面的创建与显示的。

#### 1. widget.h 文件

widget.h 文件是窗体类的头文件。在创建项目时，选择窗体基类是 QWidget，在 widget.h 中定义了一个继承自 QWidget 的类 Widget，下面是 widget.h 文件的内容。

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>

namespace Ui { //一个命名空间 Ui，包含一个类 Widget
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT
public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();
private:
    Ui::Widget *ui; //使用 Ui::Widget 定义的一个指针
};
#endif
```

widget.h 文件有几个重要的部分。

#### (1) namespace 声明

代码中有如下的一个 namespace 声明：



```
namespace Ui {
class Widget;
}
```

这是声明了一个名称为 Ui 的命名空间 (namespace)，包含一个类 Widget。但是这个类 Widget 并不是本文件里定义的类 Widget，而是 ui\_widget.h 文件里定义的类，用于描述界面组件的。这个声明相当于一个外部类型声明（具体要看完 ui\_widget.h 文件内的解释之后才能搞明白）。

(2) Widget 类的定义。widget.h 文件的主体部分是一个继承于 QWidget 的类 Widget 的定义，也就是本实例的窗体类。

在 Widget 类中使用了宏 Q\_OBJECT，这是使用 Qt 的信号与槽 (signal 和 slot) 机制的类都必须加入的一个宏（信号与槽在后面详细介绍）。

在 public 部分定义了 Widget 类的构造函数和析构函数。

在 private 部分又定义了一个指针。

```
Ui::Widget *ui;
```

这个指针是用前面声明的 namespace Ui 里的 Widget 类定义的，所以指针 ui 是指向可视化设计的界面，后面会看到要访问界面上的组件，都需要通过这个指针 ui。

## 2. widget.cpp 文件

widget.cpp 文件是类 Widget 的实现代码，下面是 widget.cpp 文件的内容。

```
#include "widget.h"
#include "ui_widget.h"
Widget::Widget(QWidget *parent) :    QWidget(parent),    ui(new Ui::Widget)
{
    ui->setupUi(this);
}
Widget::~Widget()
{
    delete ui;
}
```

注意到，在这个文件的包含文件部分自动加入了如下一行内容：

```
#include "ui_widget.h"
```

这个就是 Qt 编译生成的与 UI 文件 widget.ui 对应的类定义文件。

目前只有构造函数和析构函数。其中构造函数头部是：

```
Widget::Widget(QWidget *parent) :    QWidget(parent),    ui(new Ui::Widget)
```

其意义是：执行父类 QWidget 的构造函数，创建一个 Ui::Widget 类的对象 ui。这个 ui 就是 Widget 的 private 部分定义的指针变量 ui。

构造函数里只有一行语句：

```
ui->setupUi(this)
```

它是执行了 Ui::Widget 类的 setupUi() 函数，这个函数实现窗口的生成与各种属性的设置、信号与槽的关联（后面会具体介绍）。

析构函数只是简单地删除用 new 创建的指针 ui。



所以，在 `ui_widget.h` 文件里有一个 `namespace` 名称为 `Ui`，里面有一个类 `Widget` 是用于描述可视化设计的窗体，且与 `widget.h` 里定义类同名。在 `Widget` 类里访问 `Ui::Widget` 类的成员变量或函数需要通过 `Widget` 类里的 `ui` 指针，如同构造函数里执行 `ui->setupUi( this)` 函数那样。

### 3. `widget.ui` 文件

`widget.ui` 是窗体界面定义文件，是一个 XML 文件，定义了窗口上的所有组件的属性设置、布局，及其信号与槽函数的关联等。用 UI 设计器可视化设计的界面都由 Qt 自动解析，并以 XML 文件的形式保存下来。在设计界面时，只需在 UI 设计器里进行可视化设计即可，而不用管 `widget.ui` 文件是怎么生成的。

下面是 `widget.ui` 文件的内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
  <class>Widget</class>
  <widget class="QWidget" name="Widget">
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>336</width>
        <height>216</height>
      </rect>
    </property>
    <property name="windowTitle">
      <string>My First Demo</string>
    </property>
    <widget class="QLabel" name="LabDemo">
      <property name="geometry">
        <rect>
          <x>60</x>
          <y>50</y>
          <width>211</width>
          <height>51</height>
        </rect>
      </property>
      <property name="font">
        <font>
          <pointsize>20</pointsize>
          <weight>75</weight>
          <bold>true</bold>
        </font>
      </property>
      <property name="text">
        <string>Hello, World</string>
      </property>
    </widget>
    <widget class="QPushButton" name="btnClose">
      <property name="geometry">
        <rect>
          <x>200</x>
          <y>140</y>
          <width>81</width>
          <height>31</height>
```



```

    </rect>
</property>
<property name="text">
    <string>Close</string>
</property>
</widget>
</widget>
<layoutdefault spacing="6" margin="11"/>
<resources/>
<connections>
    <connection>
        <sender>btnClose</sender>
        <signal>clicked()</signal>
        <receiver>Widget</receiver>
        <slot>close()</slot>
    <hints>
        <hint type="sourcelabel">
            <x>240</x>
            <y>195</y>
        </hint>
        <hint type="destinationlabel">
            <x>199</x>
            <y>149</y>
        </hint>
    </hints>
    </connection>
</connections>
</ui>

```

#### 4. ui\_widget.h 文件

ui\_widget.h 是在对 widget.ui 文件编译后生成的一个文件，ui\_widget.h 会出现在编译后的目录下，或与 widget.ui 同目录（与项目的 shadow build 编译设置有关）。

文件 ui\_widget.h 并不会出现在 Qt Creator 的项目文件目录树里，当然，可以手工将 ui\_widget.h 添加到项目中。方法是在项目文件目录树上，右击项目名称节点，在调出的快捷菜单中选择“Add Existing Files…”，找到并添加 ui\_widget.h 文件即可。

---

**注意** ui\_widget.h 是对 widget.ui 文件编译后自动生成的，widget.ui 又是通过 UI 设计器可视化设计生成的。所以，对 ui\_widget.h 手工进行修改没有什么意义，所有涉及界面的修改都应该直接在 UI 设计器里进行。所以，ui\_widget.h 也没有必要添加到项目里。

---

下面是 ui\_widget.h 文件的内容。

```

/*****
** Form generated from reading UI file 'widget.ui'
** Created by: Qt User Interface Compiler version 5.9.0
** WARNING! All changes made in this file will be lost when recompiling UI file!
*****/
#ifndef UI_WIDGET_H
#define UI_WIDGET_H

#include <QtCore/QVariant>
#include <QtWidgets/QAction>
#include <QtWidgets/QApplication>

```

```

#include <QtWidgets/QButtonGroup>
#include <QtWidgets/QHeaderView>
#include <QtWidgets/QLabel>
#include <QtWidgets/QPushButton>
#include <QtWidgets/QWidget>

Qt_BEGIN_NAMESPACE
class Ui_Widget
{
public:
    QLabel *LabDemo;
    QPushButton *btnClose;

    void setupUi(QWidget *Widget)
    {
        if (Widget->objectName().isEmpty())
            Widget->setObjectName(QStringLiteral("Widget"));
        Widget->resize(280, 168);
        LabDemo = new QLabel(Widget);
        LabDemo->setObjectName(QStringLiteral("LabDemo"));
        LabDemo->setGeometry(QRect(50, 20, 201, 51));
        QFont font;
        font.setPointSize(20);
        font.setBold(true);
        font.setWeight(75);
        LabDemo->setFont(font);
        btnClose = new QPushButton(Widget);
        btnClose->setObjectName(QStringLiteral("btnClose"));
        btnClose->setGeometry(QRect(150, 120, 75, 23));

        retranslateUi(Widget);
        QObject::connect(btnClose, SIGNAL(clicked()), Widget, SLOT(close()));
        QMetaObject::connectSlotsByName(Widget);
    } // setupUi
    void retranslateUi(QWidget *Widget)
    {
        Widget->setWindowTitle(QApplication::translate("Widget", "My First Demo", Q_NULLPTR));
        LabDemo->setText(QApplication::translate("Widget", "Hello, World", Q_NULLPTR));
        btnClose->setText(QApplication::translate("Widget", "Close", Q_NULLPTR));
    } // retranslateUi
};

namespace Ui {
    class Widget: public Ui_Widget {};
} // namespace Ui
Qt_END_NAMESPACE
#endif // UI_WIDGET_H

```

查看 `ui_widget.h` 文件的内容，发现它主要做了以下的一些工作。

(1) 定义了一个类 `Ui_Widget`，用于封装可视化设计的界面。

(2) 自动生成了界面各个组件的类成员变量定义。在 `public` 部分为界面上每个组件定义了一个指针变量，变量的名称就是设置的 `objectName`。比如，在窗体上放置了一个 `QLabel` 和一个 `QPushButton` 并命名后，自动生成的定义是：

```
QLabel *LabDemo;
```

```
QPushButton *btnClose;
```

(3) 定义了 `setupUi()` 函数, 这个函数用于创建各个界面组件, 并设置其位置、大小、文字内容、字体等属性, 设置信号与槽的关联。

`setupUi()` 函数体的第一部分是根据可视化设计的界面内容, 用 C++ 代码创建界面上各组件, 并设置其属性。

接下来, `setupUi()` 调用了函数 `retranslateUi(Widget)`, 用来设置界面各组件的文字内容属性, 如标签的文字、按键的文字、窗体的标题等。将界面上的文字设置的内容独立出来作为一个函数 `retranslateUi()`, 在设计多语言界面时会用到这个函数。

`setupUi()` 函数的第三部分是设置信号与槽的关联, 本文件中有以下两行:

```
QObject::connect(btnClose, SIGNAL(clicked()), Widget, SLOT(close()));  
QMetaObject::connectSlotsByName(Widget);
```

第 1 行是调用 `connect()` 函数, 将在 UI 设计器里设置的信号与槽的关联转换为语句。这里是将 `btnClose` 按键的 `clicked()` 信号与窗体 `Widget` 的 `close()` 槽函数关联起来, 就是在图 2-4 中设置的信号与槽的关联的程序语句实现。这样, 当单击 `btnClose` 按钮时, 就会执行 `Widget` 的 `close()` 槽函数, 而 `close()` 槽函数的功能是关闭窗口。

第 2 行是设置槽函数的关联方式, 用于将 UI 设计器自动生成的组件信号的槽函数与组件信号相关联。

所以, 在 `Widget` 的构造函数里调用 `ui->setupUI(this)`, 就实现了窗体上组件的创建、属性设置、信号与槽的关联。

(4) 定义 namespace `Ui`, 并定义一个从 `Ui_Widget` 继承的类 `Widget`。

```
namespace Ui {  
    class Widget: public Ui_Widget {};  
}
```

---

提示 `ui_widget.h` 文件里实现界面功能的类是 `Ui_Widget`。再定义一个类 `Widget` 从 `Ui_Widget` 继承而来, 并定义在 namespace `Ui` 里, 这样 `Ui::Widget` 与 `widget.h` 里的类 `Widget` 同名, 但是用 namespace 区分开来。所以, 界面的 `Ui::Widget` 类与文件 `widget.h` 里定义的 `Widget` 类实际上是两个类, 但是 Qt 的处理让用户感觉不到 `Ui::Widget` 类的存在, 只需要知道在 `Widget` 类里用 `ui` 指针可以访问可视化设计的界面组件就可以了。

---

## 2.2 可视化 UI 设计

在上一节, 通过一个简单的应用程序, 分析了 Qt 创建的 GUI 应用程序中各个文件的作用, 剖析了可视化设计的 UI 文件是如何被转换为 C++ 的类定义, 并自动创建界面的。这些是使用 Qt Creator 可视化设计用户界面, 并使各个部分融合起来运行的基本原理。

本节再以一个稍微复杂的例子来讲解设计 GUI 的常见功能, 包括界面设计时布局的管理, 程序里如何访问界面组件, 以及 Qt 关键的信号与槽的概念。



## 2.2.1 实例程序功能

创建一个 Widget Application 项目 samp2\_2, 在创建窗体时选择基类 QDialog, 生成的类命名为 QWDialog, 并选择生成窗体。

如此新建的项目 samp2\_2 有一个界面文件 qwdialog.ui, 一个头文件 qwdialog.h 和源程序文件 qwdialog.cpp。此外, 还有项目文件 samp2\_2.pro 和主程序文件 main.cpp。

qwdialog.ui 界面文件设计时界面如图 2-6 所示。程序的主要功能是对中间一个文本框的文字字体样式和颜色进行设置。

在界面设计时, 对需要访问的组件修改其 objectName, 如各个按钮、需要读取输入的编辑框、需要显示结果的标签等, 以便在程序里区分。对于不需要程序访问的组件则无需修改其 objectName, 如用于界面上组件分组的 GroupBox、Frame、布局等, 让 UI 设计器自动命名即可。

对图 2-6 中几个主要组件的命名、属性设置见表 2-3。



图 2-6 实例程序 samp2\_2 设计时界面

表 2-3 qwdialog.ui 中各个组件的相关设置

对象名	类名称	属性设置	备注
txtEdit	QPlainTextEdit	Text="Hello, World It is my demo." Font.PointSize=20	用于显示文字内容, 可编辑
chkBoxUnder	QCheckBox	Text="Underline"	设置字体为下划线
chkBoxItalic	QCheckBox	Text="Italic"	设置字体为斜体
chkBoxBold	QCheckBox	Text="Bold"	设置字体为粗体
rBtnBlack	QRadioButton	Text="Black"	字体颜色为黑色
rBtnRed	QRadioButton	Text="Red"	字体颜色为红色
rBtnBlue	QRadioButton	Text="Blue"	字体颜色为蓝色
btnOK	QPushButton	Text="确定"	返回确定, 并关闭窗口
btnCancel	QPushButton	Text="取消"	返回取消, 并关闭窗口
btnClose	QPushButton	Text="退出"	退出程序
QWDialog	QWDialog	windowTitle="Dialog by Designer"	界面窗口的类名称是 QWDialog, objectName 不要修改

对于界面组件的属性设置, 需要注意以下几点。

(1) objectName 是窗体上创建的组件的实例名称, 界面上的每个组件需要有一个唯一的 objectName, 程序里访问界面组件时都是通过其 objectName 进行访问, 自动生成的槽函数名称里也有 objectName。所以, 组件的 objectName 需要在设计程序之前设置好, 设置好之后一般不要再改动。若设计程序之后再改动 objectName, 涉及的代码需要相应的改动。

(2) 窗体的 objectName 就是窗体的类名称, 在 UI 设计器里不要修改窗体的 objectName, 窗体的实例名称需要在使用窗体的代码里去定义。

## 2.2.2 界面组件布局

Qt 的界面设计使用了布局 (Layout) 功能。所谓布局, 就是界面上组件的排列方式, 使用布

局可以使组件有规则地分布，并且随着窗体大小变化自动地调整大小和相对位置。布局管理是 GUI 设计的必备技巧，下面逐步讲解如何实现图 2-6 所示的界面设计。

### 1. 界面组件的层次关系

为了将界面上的各个组件的分布设计得更加美观，经常使用一些容器类，如 `QGroupBox`、`QtabWidget`、`QFrame` 等。例如，将 3 个 `CheckBox` 组件放置在一个 `GroupBox` 组件里，该 `GroupBox` 组件就是这 3 个 `CheckBox` 的容器，移动这个 `GroupBox` 就会同时移动其中的 3 个 `CheckBox`。

图 2-7 显示的是设计图 2-6 界面的前期阶段。在窗体上放置了 2 个 `GroupBox` 组件，在 `groupBox1` 里放置 3 个 `CheckBox` 组件，在 `groupBox2` 里放置 3 个 `RadioButton` 组件。图 2-7 右侧 Object Inspector 里显示了界面上各组件之间的层次关系。

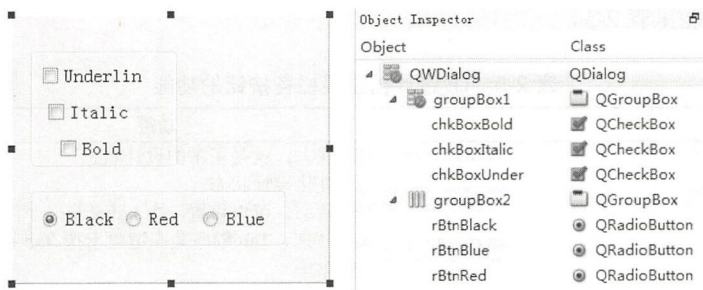


图 2-7 界面组件的放置及层次关系

### 2. 布局管理

Qt 为界面设计提供了丰富的布局管理功能，在 UI 设计器中，组件面板里有 `Layouts` 和 `Spacers` 两个组件面板，在窗体上方的工具栏里有布局管理的按钮（如图 2-8 所示）。

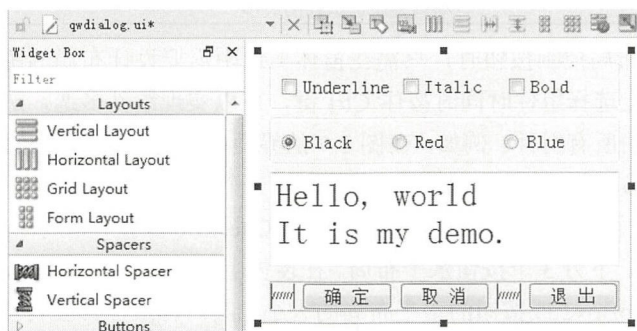





图 2-8 用于布局可视化设计的组件面板和工具栏

`Layouts` 和 `Spacers` 两个组件面板里的布局组件的功能见表 2-4。

表 2-4 组件面板上用于布局的组件

布局组件	功能
Vertical Layout	垂直方向布局，组件自动在垂直方向上分布
Horizontal Layout	水平方向布局，组件自动在水平方向上分布
Grid Layout	网格状布局，网格布局大小改变时，每个网格的大小都改变










续表

布局组件	功能
 Form Layout	窗体布局, 与网格状布局类似, 但是只有最右侧的一列网格会改变大小
 Horizontal Spacer	一个用于水平分隔的空格
 Vertical Spacer	一个用于垂直分隔的空格

使用组件面板里的布局组件设计布局时, 先拖放一个布局组件到窗体上, 如在设计图 2-8 中 3 个按钮的布局时, 先放一个 Horizontal Layout 到窗体上, 布局组件会以红色边框显示。再往布局组件里拖放 3 个 Push Button 和 2 个 Horizontal Spacer, 就可以得到图 2-8 中 3 个按钮的水平布局效果。

在设计窗体的上方有一个工具栏, 用于调整设计器进入不同的状态, 以及进行布局设计, 工具栏上各按钮的功能见表 2-5。

表 2-5 UI 设计器工具栏各按钮的功能

按钮及快捷键	功能
 Edit Widget (F3)	界面设计进入编辑状态, 就是正常的设计状态
 Edit Signals/Slots (F4)	进入信号与槽的可视化设计状态
 Edit Buddies	进入伙伴关系编辑状态, 可以设置一个 Label 与一个组件成为伙伴关系
 Edit Tab Order	进入 Tab 顺序编辑状态, Tab 顺序是在键盘上按 Tab 键时, 输入焦点在界面各组件之间跳动的顺序
 Lay Out Horizontally (Ctrl+H)	将窗体上所选组件水平布局
 Lay Out Vertically (Ctrl+L)	将窗体上所选组件垂直布局
 Lay Out Horizontally in Splitter	将窗体上所选组件用一个分割条进行水平分割布局
 Lay Out Vertically in Splitter	将窗体上所选组件用一个分割条进行垂直分割布局
 Lay Out in a Form Layout	将窗体上所选组件按窗体布局
 Lay Out in a Grid	将窗体上所选组件网格布局
 Break Layout	解除窗体上所选组件的布局, 也就是打散现有的布局
 Adjust Size (Ctrl+J)	自动调整所选组件的大小

使用工具栏上的布局控制按钮时, 只需在窗体上选中需要设计布局的组件, 然后点击某个布局按钮即可。在窗体上选择组件时同时按住 Ctrl 键, 可以实现组件多选, 选择某个容器类组件, 相当于选择了其内部的所有组件。例如, 在图 2-7 的界面中, 选中 groupBox1, 然后单击“Lay Out Horizontally”工具栏按钮, 就可以对 groupBox1 内的 3 个 CheckBox 水平布局。

在图 2-8 的界面上, 使 groupBox1 里的 3 个 CheckBox 水平布局, groupBox2 里的 3 个 RadioButton 水平布局, 下方 3 个按钮水平布局。在窗体上又放置了一个 PlainTextEdit 组件。现在, 改变 groupBox1、groupBox2 或按钮的水平布局的大小, 其内部组件都会自动改变大小。但是当改变窗体大小时, 界面上的各组件却并不会自动改变大小。

随后还需为窗体指定一个总的布局。选中窗体 (即不要选择任何组件), 单击工具栏上的“Lay Out Vertically”按钮, 使 4 个组件垂直分布。这样布局后, 当窗体大小改变时, 各个组件都会自动改变大小。

在 UI 设计器里可视化设计布局时, 要善于利用水平和垂直空格组件, 善于设置组件的最大、最小宽度和高度来实现某些需要的布局效果。

### 3. 伙伴关系与 Tab 顺序

在 UI 设计工具栏上单击“Edit Buddies”按钮可以进入伙伴关系编辑状态, 如设计一个窗体



时，进入伙伴编辑状态之后的界面如图 2-9 所示。

伙伴关系（Buddy）是指界面上一个 Label 和一个组件相关联，如图 2-9 中的伙伴关系编辑状态，单击一个 Label，按住鼠标左键，然后拖向一个组件，就建立了 Label 和组件之间的伙伴关系。

伙伴关系是为了在程序运行时，在窗体上用快捷键快速将输入焦点切换到某个组件上。例如，在图 2-9 的界面上，设定“姓名”标签的 Text 属性为“姓名(&N)”，其中符号“&”用来指定快捷字符，界面上并不显示“&”，这里指定快捷字母为 N。那么程序运行时，用户按下 Alt+N，输入焦点就会快速切换到“姓名”关联的输入框内。

在 UI 设计器工具栏上单击“Edit Tab Order”按钮进入 Tab 顺序编辑状态（如图 2-10 所示）。Tab 顺序是指在程序运行时，按下键盘上的 Tab 键时输入焦点的移动顺序。一个好的用户界面，在按 Tab 键时，焦点应该以合理的顺序在界面上移动，而不是随意地移动。

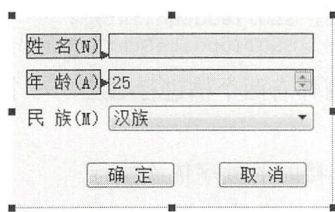


图 2-9 编辑伙伴关系

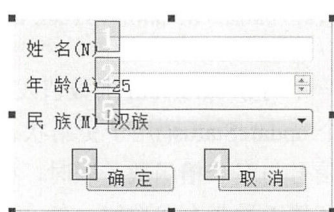


图 2-10 Tab 顺序编辑状态

进入 Tab 顺序编辑状态后，在界面上会显示具有 Tab 顺序组件的编号，依次按希望的顺序单击组件，就可以重排 Tab 顺序了。没有输入焦点的组件是没有 Tab 顺序的，如 Label 组件。

### 2.2.3 信号与槽

信号与槽（Signal & Slot）是 Qt 编程的基础，也是 Qt 的一大创新。因为有了信号与槽的编程机制，在 Qt 中处理界面各个组件的交互操作时变得更加直观和简单。

信号（Signal）就是在特定情况下被发射的事件，例如 PushButton 最常见的信号就是鼠标单击时发射的 clicked() 信号，一个 ComboBox 最常见的信号是选择的列表项变化时发射的 CurrentIndexChanged() 信号。GUI 程序设计的主要内容就是对界面上各组件的信号响应，只需要知道什么情况下发射哪些信号，合理地去响应和处理这些信号就可以了。

槽（Slot）就是对信号响应的函数。槽就是一个函数，与一般的 C++ 函数是一样的，可以定义在类的任何部分（public、private 或 protected），可以具有任何参数，也可以被直接调用。槽函数与一般的函数不同的是：槽函数可以与一个信号关联，当信号被发射时，关联的槽函数被自动执行。

信号与槽关联是用 QObject::connect() 函数实现的，其基本格式是：

```
QObject::connect(sender, SIGNAL(signal()), receiver, SLOT(slot()));
```

connect() 是 QObject 类的一个静态函数，而 QObject 是所有 Qt 类的基类，在实际调用时可以忽略前面的限定符，所以可以直接写为：

```
connect(sender, SIGNAL(signal()), receiver, SLOT(slot()));
```

其中, sender 是发射信号的对象名称, signal()是信号名称。信号可以看做是特殊的函数,需要带括号,有参数时还需要指明参数。receiver 是接收信号的对象名称, slot()是槽函数的名称,需要带括号,有参数时还需要指明参数。

SIGNAL 和 SLOT 是 Qt 的宏,用于指明信号和槽,并将它们的参数转换为相应的字符串。例如,在 samp2\_1 的 ui\_widget.h 文件中,在 setupUi()函数中有如下的语句:

```
QObject::connect(btnClose, SIGNAL(clicked()), Widget, SLOT(close()));
```

其作用就是将 btnClose 按钮的 clicked()信号与窗体(Widget)的槽函数 close()相关联,这样,当单击 btnClose 按钮(就是界面上的“Close”按钮)时,就会执行 Widget 的 close()槽函数。

关于信号与槽的使用,有以下一些规则需要注意。

(1) 一个信号可以连接多个槽,例如:

```
connect(spinNum, SIGNAL(valueChanged(int)), this, SLOT(addFun(int));
connect(spinNum, SIGNAL(valueChanged(int)), this, SLOT(updateStatus(int));
```

这是当一个对象 spinNum 的数值发生变化时,所在窗体有两个槽进行响应,一个 addFun()用于计算,一个 updateStatus()用于更新状态。

当一个信号与多个槽函数关联时,槽函数按照建立连接时的顺序依次执行。

当信号和槽函数带有参数时,在 connect()函数里,要写明参数的类型,但可以不用写参数名称。

(2) 多个信号可以连接同一个槽,例如在本项目的设计中,让三个选择颜色的 RadioButton 的 clicked()信号关联到相同的一个自定义槽函数 setTextFontColor()。

```
connect(ui->rBtnBlue, SIGNAL(clicked()), this, SLOT(setTextFontColor()));
connect(ui->rBtnRed, SIGNAL(clicked()), this, SLOT(setTextFontColor()));
connect(ui->rBtnBlack, SIGNAL(clicked()), this, SLOT(setTextFontColor()));
```

这样,当任何一个 RadioButton 被单击时,都会执行 setTextFontColor()函数。

(3) 一个信号可以连接另外一个信号,例如:

```
connect(spinNum, SIGNAL(valueChanged(int)), this, SIGNAL(refreshInfo(int));
```

这样,当一个信号发射时,也会发射另外一个信号,实现某些特殊的功能。

(4) 严格的情况下,信号与槽的参数个数和类型需要一致,至少信号的参数不能少于槽的参数。如果不匹配,会出现编译错误或运行错误。

(5) 在使用信号与槽的类中,必须在类的定义中加入宏 Q\_OBJECT。

(6) 当一个信号被发射时,与其关联的槽函数通常被立即执行,就像正常调用一个函数一样。只有当信号关联的所有槽函数执行完毕后,才会执行发射信号处后面的代码。

信号与槽机制是 Qt GUI 编程的基础,使用信号与槽机制可以比较容易地将信号与响应代码关联起来。

## 2.2.4 可视化生成槽函数原型和框架

下面开始设计程序功能。对于该程序,希望它的功能如下。

- 单击 UnderLine、Italic、Bold 3 个 CheckBox 时,根据其状态,设置 PlainTextEdit 里的文

字的字体样式;

- Black、Red、Blue 3 个 RadioButton 是互斥选择的, 单击某个 RadioButton 时, 设置文字的颜色;
- 单击“确定”“取消”或“退出”按钮时, 关闭窗口, 退出程序。

### 1. 字体样式设置

窗体在设计模式下, 选中 chkBoxUnder 组件, 单击右键调出其快捷菜单。在快捷菜单中单击菜单项“Go to slot...”, 出现如图 2-11 所示的对话框。

该对话框列出了 QCheckBox 类的所有信号, 第一个是 clicked(), 第二个是带一个布尔类型参数的 clicked(bool)。

信号 clicked(bool)会将 CheckBox 组件当前的选择状态作为一个参数传递, 在响应代码里可以直接利用这个传递的参数。而如果用信号 clicked(), 则需要在代码里读取 CheckBox 组件的选中状态。为了简化代码, 选择 clicked(bool)信号。

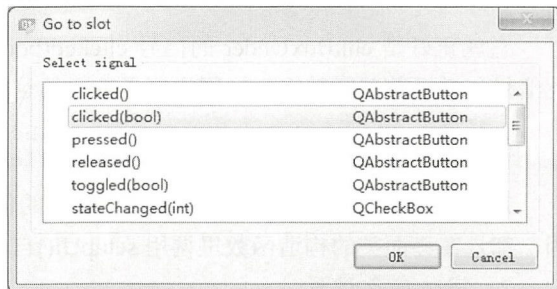


图 2-11 QCheckBox 的 Go to slot 对话框

选择 clicked(bool), 然后单击“OK”按钮, 在 QWDialog 的类定义中, 会在 private slots 部分自动增加一个槽函数声明, 函数名是根据发射对象及其信号名称自动命名的。

```
void on_chkBoxUnder_clicked(bool checked);
```

同时, 在 qwdialog.cpp 文件中自动添加了函数 on\_chkBoxUnder\_clicked(bool)的框架, 在此函数中添加如下的代码, 实现文本框字体下划线的控制。

```
void QWDialog::on_chkBoxUnder_clicked(bool checked)
{
    QFont font=ui->txtEdit->font();
    font.setUnderline(checked);
    ui->txtEdit->setFont(font);
}
```

以同样的方法为 Italic 和 Bold 两个 CheckBox 设计槽函数, 编译后运行, 发现已经实现了修改字体的下划线、斜体、粗体属性的功能, 说明信号与槽函数已经关联了。

但是, 查看 QWDialog 的构造函数, 构造函数只有简单的一条语句。

```
QWDialog::QWDialog(QWidget *parent) : QDialog(parent), ui(new Ui::QWDialog)
{
    ui->setupUi(this);
}
```

这里没有发现用 connect()函数进行几个 CheckBox 的信号与槽函数关联的操作。这些功能是如何实现的呢?

查看编译生成的 ui\_qwdialog.h 文件。构造函数里调用的 setupUi()是在 ui\_qwdialog.h 文件里实现的。查看 setupUi()函数的内容, 也没有发现用 connect()函数进行几个 CheckBox 的信号与槽关



联的操作，只是在 `setupUi()` 里发现了如下的一条语句：

```
QMetaObject::connectSlotsByName(QWDialog);
```

秘密就在于这条语句。`connectSlotsByName(QWDialog)` 函数将搜索 `QWDialog` 界面上的所有组件，将信号与槽函数匹配的信号和槽关联起来，它假设槽函数的名称是：

```
void on_<object name>_<signal name>(<signal parameters>);
```

例如，通过 UI 设计器的操作，为 `chkBoxUnder` 自动生成的槽函数是：

```
void on_chkBoxUnder_clicked(bool checked);
```

它就正好是 `chkBoxUnder` 的信号 `clicked(bool)` 的槽函数。那么，`connectSlotsByName()` 就会将此信号和槽函数关联起来，如同执行了下面的这样一条语句：

```
connect(chkBoxUnder, SIGNAL(clicked (bool)),
        this, SLOT (on_chkBoxUnder_clicked (bool)));
```

这就是用 UI 设计器可视化设计某个组件的信号响应槽函数，而不用手工去将其关联起来的原因，都是在界面类的构造函数里调用 `setupUi()` 自动完成了关联。

## 2. 字体颜色设置

设置字体的 3 个 `RadioButton` 是互斥性选择的，即一次只有一个 `RadioButton` 被选中，虽然也可以采用可视化设计的方式设计其 `clicked()` 信号的槽函数，但是这样就需要生成 3 个槽函数。这里可以简化设计，即设计一个槽函数，将 3 个 `RadioButton` 的 `clicked()` 信号关联到这一个槽函数。

为此，在 `QWDialog` 类的 `private slots` 部分增加一个槽函数定义如下：

```
void setTextFontColor();
```

---

**提示** 将鼠标光标移动到这个函数的函数名上面，单击右键，在弹出的快捷菜单中选择“Refactor”→“Add Definition in qwdialog.cpp”，就可以在 `qwdialog.cpp` 文件中自动为函数 `setTextFontColor()` 生成一个函数框架。

---

在 `qwdialog.cpp` 文件中，为 `setTextFontColor()` 编写实现代码如下：

```
void QWDialog::setTextFontColor()
{
    QPalette plet=ui->txtEdit->palette();
    if (ui->rBtnBlue->isChecked())
        plet.setColor(QPalette::Text,Qt::blue);
    else if (ui->rBtnRed->isChecked())
        plet.setColor(QPalette::Text,Qt::red);
    else if (ui->rBtnBlack->isChecked())
        plet.setColor(QPalette::Text,Qt::black);
    else
        plet.setColor(QPalette::Text,Qt::black);
    ui->txtEdit->setPalette(plet);
}
```

由于这个槽函数是自定义的，所以不会自动与 `RadioButton` 的 `clicked()` 事件关联，此时编译后运行程序不会实现改变字体颜色的功能。需要在 `QWDialog` 的构造函数中手工进行关联，代码如下：

```
QWDialog::QWDialog(QWidget *parent) :    QDialog(parent),    ui(new Ui::QWDialog)
{
    ui->setupUi(this);
```

```
connect(ui->rBtnBlue, SIGNAL(clicked()), this, SLOT(setTextFontColor()));  
connect(ui->rBtnRed, SIGNAL(clicked()), this, SLOT(setTextFontColor()));  
connect(ui->rBtnBlack, SIGNAL(clicked()), this, SLOT(setTextFontColor()));  
}
```

在构造函数中将 3 个 RadioButton 的 clicked() 信号与同一个槽函数 setTextFontColor() 相关联。再编译后运行, 就可以更改文字的颜色了。

### 3. 三个按钮的功能设计

界面上还有“确定”“取消”“退出”3 个按钮, 这是在对话框中常见的按钮。“确定”表示确认选择并关闭对话框, “取消”表示取消选择并关闭对话框, “退出”则直接关闭对话框。

QWDialog 是从 QDialog 继承而来的, QDialog 提供了 accept()、reject()、close() 等槽函数来表示这三种状态, 只需将按钮的 clicked() 信号与相应槽函数关联即可。

下面采用可视化的方式, 将按钮的 clicked() 信号与这些槽函数关联起来。在 UI 设计器里, 单击上方工具栏里的“Edit Signals/Slots”按钮, 窗体进入信号与槽函数编辑状态, 如图 2-12 所示。将鼠标移动到“确定”按钮上方, 再按下鼠标左键, 移动到窗体的空白区域释放左键, 这时出现如图 2-13 所示的关联设置对话框。

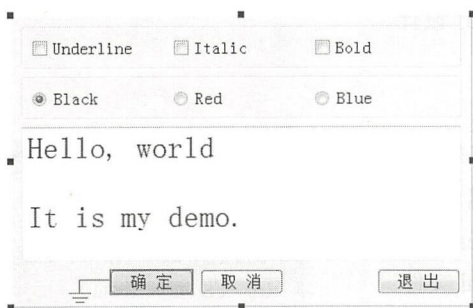


图 2-12 窗体进入 Signals/Slots 编辑状态

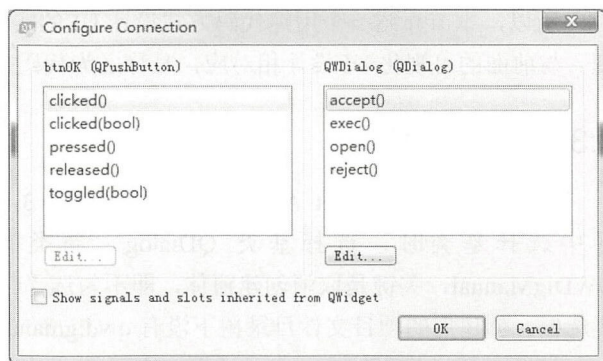


图 2-13 信号与槽关联编辑对话框

在图 2-13 中, 左侧的列表框里显示了 btnOK 的信号, 选择 clicked(), 右边的列表框里显示了 QWDialog 的槽函数, 选择 accept(), 单击“OK”按钮。

同样的方法可以将 btnCancel 的 clicked() 信号与 QWDialog 的 reject() 槽函数关联, 将 btnClose 的 clicked() 信号与 QWDialog 的 close() 槽函数关联。

**注意** 在图 2-13 的右侧列表框中没有 close() 槽函数, 需要勾选下方的“Show signals and slots inherited from QWidget”才会出现 close() 函数。

设置完 3 个按钮的信号与槽关联之后, 在窗体下方的 Signals 和 Slots 编辑器里也显示了这 3 个关联。实际上, 可以直接在 Signals 和 Slots 编辑器进行关联设置。现在编译并运行程序, 单击这 3 个按钮都会关闭程序。

那么, 这 3 个按钮的信号与槽函数的关联是在哪里实现的呢? 答案在 setupUi() 函数里, 在 setupUi() 函数里自动增加了以下 3 行代码:

```
QObject::connect(btnOK, SIGNAL(clicked()), QWDialog, SLOT(accept()));
QObject::connect(btnCancel, SIGNAL(clicked()), QWDialog, SLOT(reject()));
QObject::connect(btnClose, SIGNAL(clicked()), QWDialog, SLOT(close()));
```

这个实例程序的功能全部完成了。采用 UI 设计器设计了窗体界面，采用可视化和程序化的方式设计槽函数，设计信号与槽函数之间的关联。从以上的设计过程可以看到，Qt Creator 和 UI 设计器为设计应用程序提供了强大的可视化设计功能。

## 2.3 代码化 UI 设计

UI 的可视化设计是对用户而言的，其实底层都是 C++ 的代码实现，只是 Qt 巧妙地进行了处理，让用户省去了很多繁琐的界面设计工作。

由于界面设计的底层其实都是由 C++ 语言实现的，底层实现的功能比可视化设计更加强大和灵活。某些界面效果是可视化设计无法完成的，或者某些人习惯了用纯代码的方式来设计界面，就可以采用纯代码的方式设计界面，如 Qt 自带的实例基本都是用纯代码方式实现用户界面的。

所以，本节介绍一个用纯代码方式设计 UI 的实例，通过实例了解用纯代码设计 UI 的基本原理。与前面的可视化 UI 设计相对应，且称之为代码化 UI 设计。

### 2.3.1 实例功能

首先建立一个 Widget Application 项目 samp2\_3，在创建项目向导中选择基类时，选择基类 QDialog，新类的名称命名为 QWDlgManual，关键是取消创建窗体，即不勾选“Generate form”复选框。创建后的项目文件目录树下没有 qwdlgmanual.ui 文件。

该项目通过代码创建一个对话框，实现与 samp2\_2 类似的界面和功能。本例完成后的运行效果如图 2-14 所示，其界面和功能与 samp2\_2 类似。



图 2-14 实例 samp2\_3 运行效果

### 2.3.2 界面创建

#### 1. QWDlgManual 类定义

完成功能后的 qwdlgmanual.h 文件中 QWDlgManual 类的完整定义如下。

```
#include <QDialog>
#include <QCheckBox>
#include <QRadioButton>
#include <QPlainTextEdit>
#include <QPushButton>

class QWDlgManual : public QDialog
{
    Q_OBJECT
```



```
private:
    QCheckBox    *chkBoxUnder;
    QCheckBox    *chkBoxItalic;
    QCheckBox    *chkBoxBold;
    QRadioButton *rBtnBlack;
    QRadioButton *rBtnRed;
    QRadioButton *rBtnBlue;
    QPlainTextEdit *txtEdit;
    QPushButton  *btnOK;
    QPushButton  *btnCancel;
    QPushButton  *btnClose;
    void    iniUI(); //UI 创建与初始化
    void    iniSignalSlots(); //初始化信号与槽的链接
private slots:
    void on_chkBoxUnder(bool checked); //Underline 的槽函数
    void on_chkBoxItalic(bool checked); //Italic 的槽函数
    void on_chkBoxBold(bool checked); //Bold 的槽函数
    void setTextFontColor(); //设置字体颜色
public:
    QWDlgManual(QWidget *parent = 0);
    ~QWDlgManual();
};
```

在 QWDlgManual 类的 private 部分, 声明了界面上的各个组件的指针变量, 这些界面组件都需要在 QWDlgManual 类的构造函数里创建并在窗体上布局。

在 private 部分自定义了两个函数, iniUI()用来创建所有界面组件, 并完成布局和属性设置, iniSignalSlots()用来完成所有的信号与槽函数的关联。

在 private slots 部分声明了 4 个槽函数, 分别是 3 个 CheckBox 的响应槽函数, 以及 3 个颜色设置的 RadioButton 的共同响应槽函数。

---

**注意** 与可视化设计得到的窗体类定义不同, QWDlgManual 的类定义里没有指向界面的指针 ui。

---

这几个槽函数的功能与例 samp2\_2 中的类似, 只是在访问界面组件时, 无需使用 ui 指针, 而是直接访问 QWDlgManual 类里定义的界面组件的成员变量即可, 例如 on\_chkBoxUnder()的代码:

```
void QWDlgManual::on_chkBoxUnder(bool checked)
{
    QFont font=txtEdit->font();
    font.setUnderline(checked);
    txtEdit->setFont(font);
}
```

界面的创建, 以及信号与槽函数的关联都在 QWDlgManual 的构造函数里完成, 构造函数代码如下:

```
QWDlgManual::QWDlgManual(QWidget *parent) : QDialog(parent)
{
    iniUI(); //界面创建与布局
    iniSignalSlots(); //信号与槽的关联
    setWindowTitle("Form created manually");
}
```

构造函数调用 iniUI()创建界面组件并布局, 调用 iniSignalSlots()进行信号与槽函数的关联。

## 2. 界面组件的创建与布局

iniUI()函数实现界面组件的创建与布局, 以及属性设置。下面是 iniUI()的完整代码。

```
void QWdIlgManual::iniUI()
{ //创建 Underline, Italic, Bold 3 个 CheckBox, 并水平布局
    chkBoxUnder=new QCheckBox(tr("Underline"));
    chkBoxItalic=new QCheckBox(tr("Italic"));
    chkBoxBold=new QCheckBox(tr("Bold"));
    QHBoxLayout *HLayout1=new QHBoxLayout;
    HLayout1->addWidget(chkBoxUnder);
    HLayout1->addWidget(chkBoxItalic);
    HLayout1->addWidget(chkBoxBold);
    //创建 Black, Red, Blue 3 个 RadioButton, 并水平布局
    rBtnBlack=new QRadioButton(tr("Black"));
    rBtnBlack->setChecked(true);
    rBtnRed=new QRadioButton(tr("Red"));
    rBtnBlue=new QRadioButton(tr("Blue"));
    QHBoxLayout *HLayout2=new QHBoxLayout;
    HLayout2->addWidget(rBtnBlack);
    HLayout2->addWidget(rBtnRed);
    HLayout2->addWidget(rBtnBlue);
    //创建 确定, 取消, 退出 3 个 QPushButton, 并水平布局
    btnOK=new QPushButton(tr("确定"));
    btnCancel=new QPushButton(tr("取消"));
    btnClose=new QPushButton(tr("退出"));
    QHBoxLayout *HLayout3=new QHBoxLayout;
    HLayout3->addStretch();
    HLayout3->addWidget(btnOK);
    HLayout3->addWidget(btnCancel);
    HLayout3->addStretch();
    HLayout3->addWidget(btnClose);
    //创建 文本框, 并设置初始字体
    txtEdit=new QLineEdit;
    txtEdit->setPlainText("Hello world\n\nIt is my demo");
    QFont font=txtEdit->font(); //获取字体
    font.setPointSize(20); //修改字体大小
    txtEdit->setFont(font); //设置字体
    //创建 垂直布局, 并设置为主布局
    QVBoxLayout *VLayout=new QVBoxLayout;
    VLayout->addLayout(HLayout1); //添加字体类型组
    VLayout->addLayout(HLayout2); //添加字体颜色组
    VLayout->addWidget(txtEdit); //添加 QLineEdit
    VLayout->addLayout(HLayout3); //添加按钮组
    setLayout(VLayout); //设置为窗体的主布局
}
```

iniUI()函数按照顺序完成了如下的功能。

- 创建 3 个 QCheckBox 组件, 这 3 个组件的指针已经定义为 QWdIlgManual 的私有变量, 然后创建一个水平布局 HLayout1, 将 3 个 CheckBox 添加到这个水平布局里。
- 创建 3 个 QRadioButton 组件, 并创建一个水平布局 HLayout2, 将 3 个 RadioButton 添加到这个水平布局里。
- 创建 3 个 QPushButton 组件, 并创建一个水平布局 HLayout3, 将 3 个 QPushButton 添加到这个

水平布局里，并适当添加水平空格。

- 创建一个 QPlainTextEdit 组件，设置其文字内容，并设置其字体。
- 创建一个垂直布局 VLayout，将前面创建的 3 个水平布局和文本框依次添加到此布局里。
- 设置垂直布局为窗体的主布局。

如此创建组件并设置布局后，运行可以得到如图 2-14 所示的界面效果。这里完全是采用代码来实现组件创建与布局的设置，而这些功能在可视化设计中是由 setupUi() 函数根据界面的可视化设计结果自动实现的。

采用代码设计实现 UI 时，需要对组件的布局有个完整的规划，不如可视化设计直观，且编写代码工作量大。

### 2.3.3 信号与槽的关联

在纯代码设计 UI 时，信号与槽的关联也需要用代码来完成。函数 iniSignalSlots() 初始化所有的信号与槽的关联，其完整代码如下。

```
void QWDlgManual::iniSignalSlots()
{
    //三个颜色 QPushButton 的 clicked() 信号与 setTextFontColor() 槽函数关联
    connect(rBtnBlue, SIGNAL(clicked()), this, SLOT(setTextFontColor()));
    connect(rBtnRed, SIGNAL(clicked()), this, SLOT(setTextFontColor()));
    connect(rBtnBlack, SIGNAL(clicked()), this, SLOT(setTextFontColor()));
    //三个字体设置的 QCheckBox 的 clicked(bool) 信号与相应的槽函数关联
    connect(chkBoxUnder, SIGNAL(clicked(bool)),
            this, SLOT(on_chkBoxUnder(bool)));
    connect(chkBoxItalic, SIGNAL(clicked(bool)),
            this, SLOT(on_chkBoxItalic(bool)));
    connect(chkBoxBold, SIGNAL(clicked(bool)),
            this, SLOT(on_chkBoxBold(bool)));
    //三个按钮的信号与窗体的槽函数关联
    connect(btnOK, SIGNAL(clicked()), this, SLOT(accept()));
    connect(btnCancel, SIGNAL(clicked()), this, SLOT(reject()));
    connect(btnClose, SIGNAL(clicked()), this, SLOT(close()));
}
```

设计完成后，编译并运行程序，可以得到如图 2-14 所示的运行效果，且功能与 samp2\_2 相同。很显然，采用纯代码方式实现 UI 界面是比较复杂的，代码设计的工作量大而繁琐。

## 2.4 混合方式 UI 设计

### 2.4.1 设计目的

可视化 UI 设计无需人工编写代码去处理大量繁琐的界面组件的创建和布局管理工作，可以直观地进行界面设计，大大提高工作效率。但是可视化 UI 设计也存在一些缺陷，如某些组件无法可视化地添加到界面上，比如在工具栏上无法可视化添加 ComboBox 组件，而用代码就可以。

采用纯代码方式进行 UI 设计虽然无所不能，但是设计效率太低，过程非常繁琐，而可视化



UI 设计简单高效。所以，能用可视化设计的就尽可能用可视化设计解决，无法解决的再用纯代码方式，将两种方法结合，才是高效设计 UI 的方法。

本节用一个实例讲解如何用混合方式创建 UI，即部分界面设计用 UI 设计器可视化实现，部分无法在 UI 设计器里实现的界面设计用代码实现。同时用这个实例讲解如何使用资源文件、如何使用 Actions，如何设计主窗口里的菜单、工具栏和状态栏，这些是一般应用程序主窗口都有的功能。

图 2-15 是在 UI 设计器里设计完成的窗口。本项目的窗口类是从 QMainWindow 继承而来的，具有主菜单、工具栏和状态栏。这个例子实现了主菜单、主工具栏和状态栏，中间工作区里是一个 TextEdit 组件，用于编辑文本。

我们希望在工具栏上添加一个 SpinBox 组件，用于设置字体大小，还想在工具栏上添加一个 FontComboBox 组件，用于选择字体名称。但是在 UI 设计器里，将这些组件拖放到工具栏上时，会显示不能添加到工具栏上。同样，在窗体下方的状态栏上也不能直接添加 Label 和 ProgressBar 组件。这就是 UI 可视化设计的局限，无法实现某些界面效果。

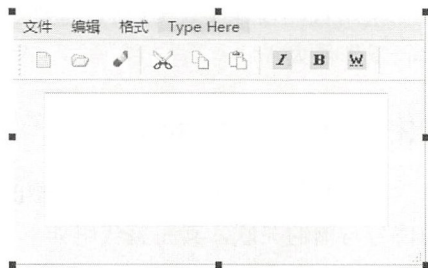


图 2-15 在 UI 设计器里完成的窗口界面

但是通过编写代码，可以实现期望的界面效果。图 2-16 就是程序运行时的窗口，可见在工具栏上添加了设置字体大小的 SpinBox 组件和选择字体名称的 FontComboBox 组件，在状态栏上添加了显示当前文件名称的标签，还添加了一个 ProgressBar。通过设计相应的槽函数并与界面组件的相关信号进行关联，实现了期望的程序功能。

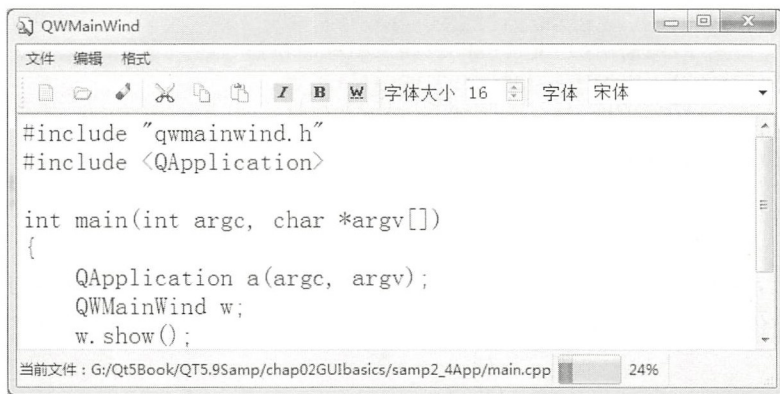


图 2-16 程序运行时的窗口界面

这就是混合方式界面设计的效果，先将所有在 UI 设计器里可设计的静态 UI 元素都用可视化的方式实现，不能在 UI 设计器里可视化设计的再用代码实现。

## 2.4.2 创建项目并添加资源文件

创建一个 Widget Application 项目，在向导的创建窗口类时，选择基类 QMainWindow，新建类的名称设置为 QWMainWind，并选择生成窗体。

本项目主窗口有菜单和工具栏，需要使用到图标。在 Qt 项目中，图标可以存储在资源文件里，为此先创建一个资源文件。在 Qt Creator 里单击“File”→“New File or Project...”菜单项，在新建文件与项目对话框里选择“Qt Resource File”，然后按照向导的指引设置资源文件的文件名，并添加到当前项目里。

本项目创建的资源文件名为 res.qrc。在项目文件目录树里，会自动创建一个与 Headers、Sources 和 Forms 并列的 Resources 文件组，在 Resources 组里有 res.qrc 节点。在资源文件名节点上右击，在弹出的快捷菜单中选择“Open in Editor”打开资源文件编辑器（如图 2-17 所示）。

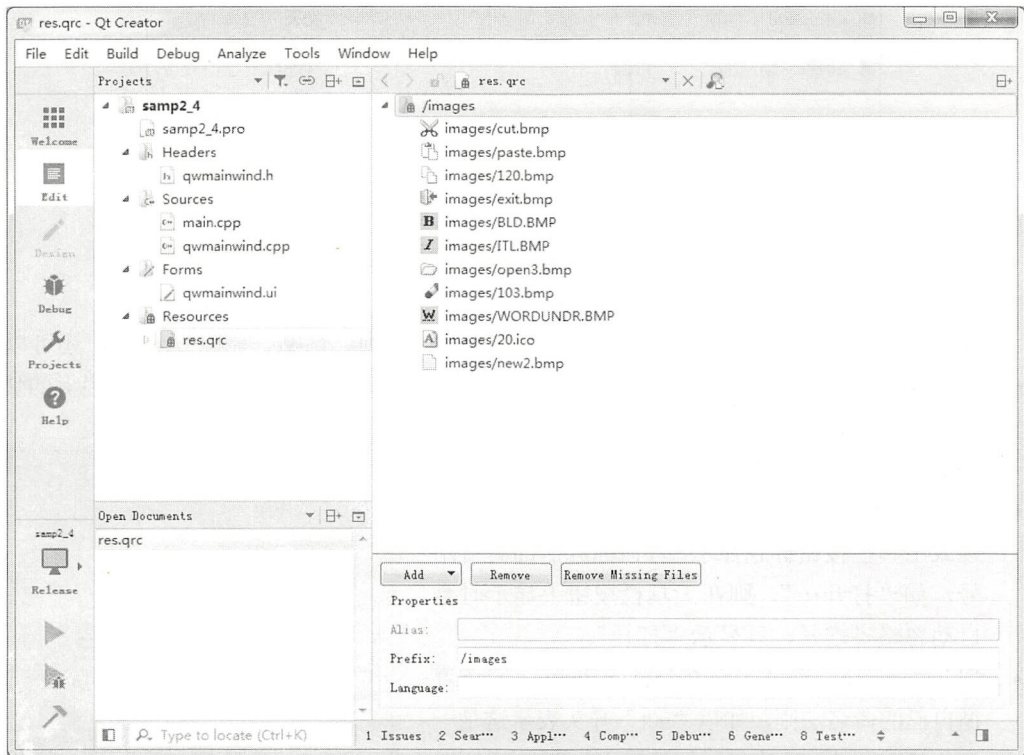


图 2-17 资源文件编辑器

资源文件最主要的一个功能就是存储图标和图片文件，以便在程序里使用。在资源文件里首先建一个前缀（Prefix），例如 images，方法是在图 2-17 显示的窗口右下方的功能区单击“Add”按钮下的“Add Prefix”，设置一个前缀名，前缀就类似于是资源的分组。然后再单击“Add”按钮下的“Add Files”选择图标文件即可。如果所选的图标文件不在本项目的子目录里，会提示复制文件到项目下的子目录。所以，最好将图标等原始文件放在项目的子目录下。

### 2.4.3 设计 Action

QAction 是一个非常有用的类，在界面设计时创建 Action，并编写其 trigger() 信号的槽函数。使用设计的 Action 可以创建菜单项、工具栏按钮，还可以设置为 QToolButton 按钮的关联 Action。

点击这些由 Action 创建的菜单项、按钮就是执行 Action 的槽函数。

在项目文件目录树里双击 qwmainwind.ui, 进入 UI 设计器, 在窗体的下方有一个 Action Editor 的面板, 图 2-18 是本项目设计好之后的 Action 列表。根据图标和文字就可以知道每个 Action 的功能。

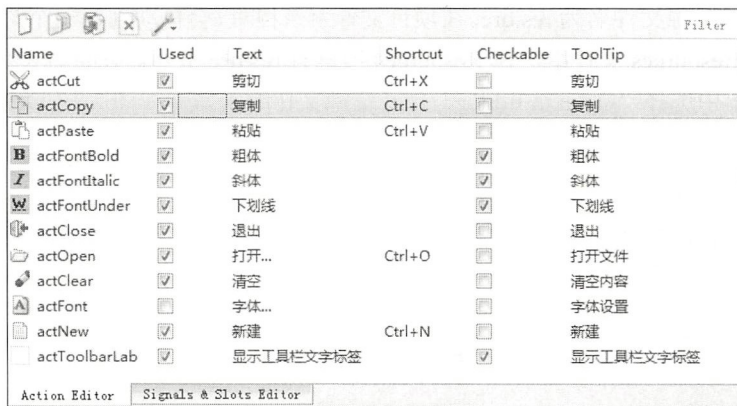


图 2-18 Action 编辑器

在 Action 编辑器的上方有一个工具栏, 可以新建、复制、粘贴、删除 Action, 还可以设置 Action 列表的显示方式。若要编辑某个 Action, 在列表里双击该 Action 即可。单击工具栏上的“New”按钮可以新建一个 Action。新建或编辑 Action 的对话框如图 2-19 所示。

在此对话框里有以下的一些设置。

- **Text:** Action 的显示文字, 该文字会作为菜单标题或工具栏按钮标题显示。若该标题后面有省略号, 如“打开...”, 则在工具栏按钮上显示时会自动忽略省略号, 只显示“打开”。
- **Object name:** 该 Action 的 objectName。应该遵循自己的命名法则, 如以“act”开头表示这是一个 Action。图 2-19 中显示的是打开文件的 Action, 命名为 actOpen。如果一个界面上 Action 比较多, 还可以分组命名, 如 actFileOpen、actFileNew 和 actFileSave 表示“文件”组, 而 actEditCut、actEditCopy、actEditPaste 等表示“编辑”组。
- **ToolTip:** 这个文字内容是当鼠标在一个菜单项或工具栏按钮上短暂停留时出现的提示文字。
- **Icon:** 设置 Action 的图标, 单击其右侧的按钮可以从资源文件里选择图标, 或者直接选择图片文件作为图标。
- **Checkable:** 设置 Action 是否可以被复选, 如果选中此选项, 那么该 Action 就类似于 QCheckbox 可以改变其复选状态。
- **Shortcut:** 设置快捷键, 将输入光标移动到 Shortcut 旁边的编辑框里, 然后按下想要设置的快捷键即可, 如“Ctrl+O”。



图 2-19 新建或编辑一个 Action



做好这些设置后,单击“OK”按钮就可以新建或修改 Action 了。所有用于菜单和工具栏设计的功能都需要用 Action 来实现。

### 2.4.4 设计菜单和工具栏

建立 Action 之后,就可以在主窗体上设计菜单和工具栏了。本项目的窗体类 QWMainWind 是从 QMainWindow 继承的,具有菜单栏、工具栏和状态栏。

双击项目文件目录树里的 qwmainwind.ui,在 UI 设计器里打开此窗口。在窗口最上方显示“Type Here”的地方是菜单栏,菜单栏下方是工具栏,窗口最下方是状态栏。

在菜单栏显示“Type Here”的地方双击,出现一个编辑框,在编辑框里输入所要设计菜单的分组名称,如“文件”,然后回车,这样就创建了一个“文件”菜单分组,同样可以创建“编辑”“格式”分组。

创建主菜单的分组后,从 Action 编辑器的列表里将一个 Action 拖放到菜单某个分组下,就可以创建一个菜单项,如同在界面上放置一个组件一样。如果需要在菜单里增加一个分隔条,双击“Add Separator”就可以创建一个分隔条,然后拖动到需要的位置即可。如果需要删除某个菜单项或分隔条,单击右键,选择“Remove”菜单项。菜单设计的结果如图 2-20 所示。



图 2-20 完成后的菜单栏各分组下的菜单项

设计工具栏的方式也很相似。将一个 Action 拖放到窗口的工具栏上,就会新建一个工具栏按钮。同样可以在工具栏上添加分隔条,可以移除工具栏按钮。主窗口上初始的只有一个工具栏,如果需要设计多个工具栏,在主窗口上单击右键,单击“Add Tool Bar”即可新建一个工具栏。

工具栏上按钮的显示方式有多种,只需设置工具栏的 `toolButtonStyle` 属性,这是 `Qt::ToolButtonStyle` 枚举类型,缺省的是 `Qt::ToolButtonIconOnly`,即只显示按钮的图标。还可以设置为:

- `Qt::ToolButtonTextBesideIcon`——文字显示在按钮旁边;
- `Qt::ToolButtonTextOnly`——只显示文字;
- `Qt::ToolButtonTextUnderIcon`——文字显示在按钮下方。

同时在窗体上放置一个 `QTextEdit` 组件,其 `objectname` 设置为 `txtEdit`。如此就完成了菜单栏、工具栏的可视化设计。

可视化设计的界面的底层实现是由 `ui_qwmainwind.h` 文件实现的。打开 `ui_qwmainwind.h` 文件,能看到如下代码(已删除部分设置布局的代码)。

```
class Ui_QWMainWind
```

```
{
public:
    QAction *actCut;
    QAction *actCopy;
    QAction *actPaste;
    QAction *actFontBold;
    QAction *actFontItalic;
    QAction *actFontUnder;
    QAction *actClose;
    QAction *actOpen;
    QAction *actClear;
    QAction *actFont;
    QAction *actNew;
    QAction *actToolBarLab;
    QWidget *centralWidget;
    QTextEdit *txtEdit;
    QSpinBox *spinBox;
    QFontComboBox *fontComboBox;
    QMenuBar *menuBar;
    QMenu *menu;
    QMenu *menu_2;
    QMenu *menu_3;
    QToolBar *mainToolBar;
    QStatusBar *statusBar;

    void setupUi(QMainWindow *QWMainWind)
    {
        if (QWMainWind->objectName().isEmpty())
            QWMainWind->setObjectName(QStringLiteral("QWMainWind"));
        QWMainWind->resize(586, 363);
        QFont font;
        font.setPointSize(11);
        QWMainWind->setFont(font);

        actCut = new QAction(QWMainWind);
        actCut->setObjectName(QStringLiteral("actCut"));
        QIcon icon;
        icon.addFile(QStringLiteral(":/images/images/cut.bmp"), QSize(), QIcon::Normal,
QIcon::Off);
        actCut->setIcon(icon);

        actCopy = new QAction(QWMainWind);
        actCopy->setObjectName(QStringLiteral("actCopy"));
        QIcon icon1;
        icon1.addFile(QStringLiteral(":/images/images/120.bmp"), QSize(), QIcon::Normal,
QIcon::Off);
        actCopy->setIcon(icon1);

        // 创建其他 action, 省略...
        centralWidget = new QWidget(QWMainWind);
        centralWidget->setObjectName(QStringLiteral("centralWidget"));
        txtEdit = new QTextEdit(centralWidget);
        txtEdit->setObjectName(QStringLiteral("txtEdit"));
        txtEdit->setGeometry(QRect(20, 20, 231, 221));
        QFont font1;
        font1.setPointSize(16);
```

```

        txtEdit->setFont(font1);
        QWMainWind->setCentralWidget(centralWidget);
//主窗体创建菜单栏、工具栏和状态栏
        menuBar = new QMenuBar(QWMainWind);
        menuBar->setObjectName(QStringLiteral("menuBar"));
        menuBar->setGeometry(QRect(0, 0, 586, 23));
        menu = new QMenu(menuBar);
        menu->setObjectName(QStringLiteral("menu"));
        menu_2 = new QMenu(menuBar);
        menu_2->setObjectName(QStringLiteral("menu_2"));
        menu_3 = new QMenu(menuBar);
        menu_3->setObjectName(QStringLiteral("menu_3"));
        QWMainWind->setMenuBar(menuBar);
        mainToolBar = new QToolBar(QWMainWind);
        mainToolBar->setObjectName(QStringLiteral("mainToolBar"));
        mainToolBar->setToolButtonStyle(Qt::ToolButtonTextUnderIcon);
        QWMainWind->addToolBar(Qt::TopToolBarArea, mainToolBar);
        statusBar = new QStatusBar(QWMainWind);
        statusBar->setObjectName(QStringLiteral("statusBar"));
        QWMainWind->setStatusBar(statusBar);

//主菜单添加菜单项
        menuBar->addAction(menu->menuAction());
        menuBar->addAction(menu_2->menuAction());
        menuBar->addAction(menu_3->menuAction());
        menu->addAction(actNew);
        menu->addAction(actOpen);
        menu->addSeparator();
        menu->addAction(actClose);
        menu_2->addAction(actCut);
        menu_2->addAction(actCopy);
        menu_2->addAction(actPaste);
        menu_2->addSeparator();
        menu_2->addAction(actClear);
        menu_3->addAction(actFontBold);
        menu_3->addAction(actFontItalic);
        menu_3->addAction(actFontUnder);
        menu_3->addSeparator();
        menu_3->addAction(actToolBarLab);
//主工具栏添加按钮
        mainToolBar->addAction(actNew);
        mainToolBar->addAction(actOpen);
        mainToolBar->addAction(actClear);
        mainToolBar->addSeparator();
        mainToolBar->addAction(actCut);
        mainToolBar->addAction(actCopy);
        mainToolBar->addAction(actPaste);
        mainToolBar->addSeparator();
        mainToolBar->addAction(actFontItalic);
        mainToolBar->addAction(actFontBold);
        mainToolBar->addAction(actFontUnder);
        mainToolBar->addSeparator();

        retranslateUi(QWMainWind);
        QMetaObject::connectSlotsByName(QWMainWind);
    } // setupUi
};

```



ui\_qmainwindow.h 中定义了类 Ui\_QMainWindow。在 public 部分定义了界面所有 Action 和各种组件的指针变量，在 setupUi() 函数里依次创建所有的 Action 和界面组件，然后将 Action 添加到主菜单和工具栏上。

ui\_qmainwindow.h 的代码很长，上面的清单已省略了若干行。查看 ui\_qmainwindow.h 里的代码有助于了解 Action、菜单、工具栏的代码实现原理。若按照实例 samp2\_3 的方法完全通过手工的方式来设计这样一个窗口无疑是一件繁琐的工作。好在采用 UI 设计器设计界面时，不需要动手去编写这些代码。界面设计的代码实现交给 Qt 去做就好了，它比手工编写代码效率高。

## 2.4.5 代码创建其他界面组件

至此，在 UI 设计器里已经设计了这个应用程序的主要界面功能。现在想要实现如图 2-16 所示的界面，在工具栏上增加一个 SpinBox 用于设置字体大小，增加一个 FontComboBox 来选择字体。当从组件面板里拖放一个 SpinBox 到工具栏上时，却发现工具栏“拒收”！同样，在状态栏上放一个 Label 和一个 ProgessBar 也是被“拒收”的。这是 UI 设计器的局限性，某些界面效果无法用可视化设计方式实现。

为此，需要编写一些代码来实现这些无法可视化设计的界面功能。先在 QMainWindow 类定义里增加一些变量和函数定义，增加的定义如下：

```
class QMainWindow : public QMainWindow
{
private:
    QLabel      *fLabCurFile; //状态栏里显示当前文件的 Label
    QProgressBar *progressBar1; //状态栏上的进度条
    QSpinBox     *spinFontSize; // 字体大小 spinBox
    QFontComboBox *comboBox; //字体名称 comboBox
    void iniUI(); //代码实现的 UI 初始化
};
```

iniUI() 函数用于创建这些界面组件，并添加到工具栏或状态栏，其实现代码如下：

```
void QMainWindow::iniUI()
{//状态栏上添加组件
    fLabCurFile=new QLabel;
    fLabCurFile->setMinimumWidth(150);
    fLabCurFile->setText("当前文件: ");
    ui->statusBar->addWidget(fLabCurFile); //添加到状态栏

    progressBar1=new QProgressBar;
    progressBar1->setMaximumWidth(200);
    progressBar1->setMinimum(5);
    progressBar1->setMaximum(50);
    progressBar1->setValue(ui->textEdit->font().pointSize());
    ui->statusBar->addWidget(progressBar1); //添加到状态栏
//工具栏上添加组件
    spinFontSize = new QSpinBox; //文字大小 SpinBox
    spinFontSize->setMinimum(5);
    spinFontSize->setMaximum(50);
    spinFontSize->setValue(ui->textEdit->font().pointSize());
    spinFontSize->setMinimumWidth(50);
```

```

ui->mainToolBar->addWidget(new QLabel("字体大小 "));
ui->mainToolBar->addWidget(spinFontSize); //SpinBox 添加到工具栏

ui->mainToolBar->addSeparator(); //分隔条
ui->mainToolBar->addWidget(new QLabel(" 字体 "));
comboFont = new QFontComboBox;
comboFont->setMinimumWidth(150);
ui->mainToolBar->addWidget(comboFont); //添加到工具栏

setCentralWidget(ui->txtEdit);
}

```

iniUI()函数实现如下的功能。

- 创建一个 QLabel 类型的组件 fLabCurFile 并将其添加到状态栏。
- 创建一个 QProgressBar 类型的组件 progressBar1 并添加到状态栏。
- 创建 QSpinBox 类型组件 spinFontSize 并设置其属性，然后添加到工具栏。
- 创建一个 QFontComboBox 类型的组件并添加到工具栏里。

实现了 iniUI()函数之后，在 QWMainWind 的构造函数里调用 iniUI()函数，现在的构造函数代码实现如下：

```

QWMainWind::QWMainWind(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::QWMainWind)
{
    ui->setupUi(this);
    iniUI();
}

```

**注意** iniUI()函数一定要在 ui->setupUi(this)之后再调用，两行语句的先后顺序不能调换。因为 ui->setupUi(this)实现了可视化设计的界面的创建，iniUI()是在可视化创建的界面基础之上添加其他组件，所以必须在其后面调用。

现在编译后运行就可以出现如图 2-17 所示的界面了，界面由可视化设计和代码设计混合实现。

## 2.4.6 Action 的功能实现

Action 是一种不可见的界面元素，主要用于菜单项、工具栏按钮的设计。Action 的主要信号是 trigger()，为一个 Action 的 trigger()信号编写槽函数之后，菜单和工具栏上由此 Action 创建的菜单项和工具栏按钮就都关联此槽函数。

### 1. 编辑功能 Action 的实现

用于编辑的 Action 有剪切、复制、粘贴和清除的功能，以便对 txtEdit 组件进行相应的操作。而 QTextEdit 类就有相应的槽函数，无需再编写实现代码，只需将 Action 的 trigger()信号与 txtEdit 的相应槽函数进行关联即可。在 Signals 和 Slots 编辑器里设置信号与槽的关联，设计好后的几个 Action 的关联如图 2-21 所示。

Sender	Signal	Receiver	Slot
actPaste	triggered()	txtEdit	paste()
actCut	triggered()	txtEdit	cut()
actCopy	triggered()	txtEdit	copy()
actClose	triggered()	QWMainWind	close()
actClear	triggered()	txtEdit	clear()

图 2-21 在信号与槽编辑器里设置关联

## 2. 其他 Action 的功能实现

Action 的主要信号是 `trigger()` 和 `trigger(bool)`，在单击菜单项或工具栏按钮时发射。

用于设置粗体、斜体和下划线的 3 个 Action 具有 `Checkable` 属性，选择 `trigger(bool)` 信号设计槽函数更合适，该信号会将 Action 的复选状态作为一个参数传递，在响应代码里可以直接使用。其他 Action 可选择 `trigger()` 信号生成槽函数。在 Action 的“Go to slot”对话框，选择信号为 Action 创建槽函数。

下面是用于设置字体粗体的 `actFontBold` 槽函数代码，使用了 `trigger(bool)` 信号。

```
void QWMainWind::on_actFontBold_triggered(bool checked)
{
    QTextCharFormat fmt;
    fmt=ui->txtEdit->currentCharFormat();
    if (checked)
        fmt.setFontWeight(QFont::Bold);
    else
        fmt.setFontWeight(QFont::Normal);
    ui->txtEdit->mergeCurrentCharFormat(fmt);
}
```

其他 Action 的槽函数代码略。本实例只是为介绍混合法 UI 设计，以及应用程序开发的基本流程和方法，程序功能的具体实现不是本例的重点。例如，“打开文件”的代码实现涉及对话框调用、文件读取、数据流等操作，这些会在后续章节里介绍。

## 3. Action 的 enabled 和 checked 属性的更新

为了使界面和程序功能显得更加智能一点，应该根据当前的状态自动更新相关 Action 的 `checked` 和 `enabled` 属性，这是软件设计中常见的功能。

在本程序中，“剪切”“复制”“粘贴”的 `enabled` 属性应该随文本框内文字选择的状态变化而变化，“粗体”“斜体”“下划线”的 `checked` 属性应该随着当前文字的字体状态而自动更新。这可以针对 `QTextEdit` 的一些信号编写槽函数来实现。

在主窗体上选择文本编辑框 `txtEdit`，在快捷菜单里调出“Go to slot”对话框。对话框里列出了 `QTextEdit` 的所有信号，有两个可以利用的信号。

- `copyAvailable(bool)` 信号在有内容可以被复制时发射，并且传递了一个布尔参数，可以利用此信号来改变 `actCut`，`actCopy` 的 `enabled` 属性。
- `selectionChanged()` 信号在选择的文字发生变化时发射，利用此信号，可以读取当前文字的格式，从而更新粗体、斜体和下划线 3 种字体设置 Action 的 `checked` 属性。

为 `txtEdit` 组件的这两个信号生成槽函数定义和函数体框架，编写代码如下：

```
void QWMainWind::on_txtEdit_copyAvailable(bool b)
{ //更新 cut,copy,paste 的 enabled 属性
    ui->actCut->setEnabled(b);
    ui->actCopy->setEnabled(b);
    ui->actPaste->setEnabled(ui->txtEdit->canPaste());
}

void QWMainWind::on_txtEdit_selectionChanged()
{ //更新粗体、斜体和下划线 3 种 action 的 checked 属性
    QTextCharFormat fmt;
```



```

fmt=ui->txtEdit->currentCharFormat(); //获取文字的格式
ui->actFontItalic->setChecked(fmt.fontItalic()); //是否斜体
ui->actFontBold->setChecked(fmt.font().bold()); //是否粗体
ui->actFontUnder->setChecked(fmt.fontUnderline()); //是否有下划线
}

```

## 2.4.7 手工创建的组件的信号与槽

在界面上还有两个用代码创建的组件，用于设置字体大小的 `spinFontSize` 和用于设置字体的 `comboFont`，这两个组件的信号与槽的功能实现显然不能通过前面所述的可视化的方法来实现。

对于手工创建的组件需要手工编写槽函数，并将信号与槽关联起来。

首先，需要确定利用组件的哪个信号来编写响应代码。例如，用于设置字体大小的 `SpinBox` 组件最合适的信号是 `valueChanged(int)`，用于字体设置的 `FontComboBox` 组件适合使用 `currentIndexChanged(QString)` 信号。

为此需要在 `QWMainWind` 类中定义两个槽函数，以及用于进行信号与槽关联的函数 `iniSignalSlots()`，该函数在构造函数里调用。下面是 `QWMainWind` 类中的相关定义（省略了其他代码）。

```

class QWMainWind : public QMainWindow
{
private:
    void iniSignalSlots(); //关联信号与槽
private slots:
// 自定义槽函数
    void on_spinBoxFontSize_valueChanged(int aFontSize); //改变字体大小
    void on_comboFont_currentIndexChanged(const QString &arg1); //选择字体
};

```

下面的代码是以上 3 个函数的实现。

```

void QWMainWind::iniSignalSlots()
{ //信号与槽的关联
    connect(spinFontSize, SIGNAL(valueChanged(int)),
            this, SLOT(on_spinBoxFontSize_valueChanged(int)));
    connect(comboFont, SIGNAL(currentIndexChanged(const QString &)),
            this, SLOT(on_comboFont_currentIndexChanged(const QString &)));
}
void QWMainWind::on_spinBoxFontSize_valueChanged(int aFontSize)
//改变字体大小的 SpinBox
{
    QTextCharFormat fmt;
    fmt.setFontPointSize(aFontSize); //设置字体大小
    ui->txtEdit->mergeCurrentCharFormat(fmt);
    progressBar1->setValue(aFontSize);
}
void QWMainWind::on_comboFont_currentIndexChanged(const QString &arg1)
//FontCombobox 的响应，选择字体名称
{
    QTextCharFormat fmt;
    fmt.setFontFamily(arg1); //设置字体名称
    ui->txtEdit->mergeCurrentCharFormat(fmt);
}

```

然后在 `QWMainWind` 的构造函数里调用 `iniSignalSlots()` 函数，实现手工创建的组件的信号与槽关联。

2.4.8 为应用程序设置图标

用 Qt Creator 创建的项目编译后的可执行文件具有默认的图标，如果需要为应用设置一个自己的图标，其操作很简单，只需两步。

- 将一个图标文件（必须是“.ico”后缀的图标文件）复制到项目源程序目录下。
- 在项目配置文件里用 RC\_ICONS 设置图标文件名，添加下面一行代码。

```
RC_ICONS = AppIcon.ico
```

其中，“AppIcon.ico”就是复制到项目源程序目录下的图标文件名称。这样设置后，编译后生成的可执行文件，以及主窗口的图标就换成设置的图标了。

至此，这个例子的全部功能就都实现了。在这个实例里，我们介绍了 Action 的创建和使用，采用可视化的方式设计了大部分界面和槽函数。又采用手工编写代码的方式添加了其他的组件到界面上，并编写槽函数，进行信号与槽函数的关联。

这种可视化与代码化混合的设计方式与纯代码方式相比，避免了创建界面时大量繁琐的创建与布局工作，可以大大提高设计效率，同时又用代码实现了一些在 UI 设计器里无法可视化完成的一些界面效果。

2.5 Qt Creator 使用技巧

Qt Creator 在设计界面或编辑代码时，有一些快捷键和使用技巧，熟悉这些快捷键和使用技巧，可以提高工作效率。表 2-6 是 Qt Creator 的一些快捷操作的总结。

表 2-6 源程序编辑器的快捷操作

功能	快捷键	解释
Switch Header/Source	F4	在同名的头文件和源程序文件之间切换
Follow Symbol Under Cursor	F2	跟踪光标下的符号，若是变量，可跟踪到变量声明的地方；若是函数体或函数声明，可在两者之间切换
Switch Between Function Declaration and Definition	Shift+F2	在函数的声明（函数原型）和定义（函数实现）之间切换
Refactor/Rename Symbol Under Cursor	Ctrl+Shift+R	对光标处的符号更改名称，这将替换到所有用到这个符号的地方
Refactor/Add Definition in .cpp		为函数原型在 cpp 文件里生成函数体
Auto-indent Selection	Ctrl+I	为选择的文字自动进行缩进
Toggle Comment Selection	Ctrl+/ F1	为选择的文字进行注释符号的切换，即可以注释所选代码，或取消注释
Context Help		为光标所在的符号显示帮助文件的内容
Save All	Ctrl+Shift+S	文件全部保存
Find/Replace	Ctrl+F	调出查找/替换对话框
Find Next	F3	查找下一个
Build	Ctrl+B	编译当前项目
Start Debugging	F5	开始调试
Step Over	F10	调试状态下单步略过，即执行当前程序语句
Step Into	F11	调试状态下跟踪进入，即如果当前行里有函数，就跟踪进入函数体
Toggle Breakpoint	F9	设置或取消当前行的断点设置

另外，在使用 Qt 时，要善于使用 Qt 自带的帮助文件，对于一个编程语言或类库来说，其自

带的帮助文件是最全面最权威的资料。当光标停留在一个类名或函数上时，按 F1 可以调出其帮助文件的内容。

在 Qt Creator 主窗口左侧的主工具栏上有“Help”按钮，单击可以打开 Qt 的帮助文件系统（如图 2-22 所示），也可以使用“开始”菜单 Qt 程序组里的 Assistant 单独打开帮助系统。

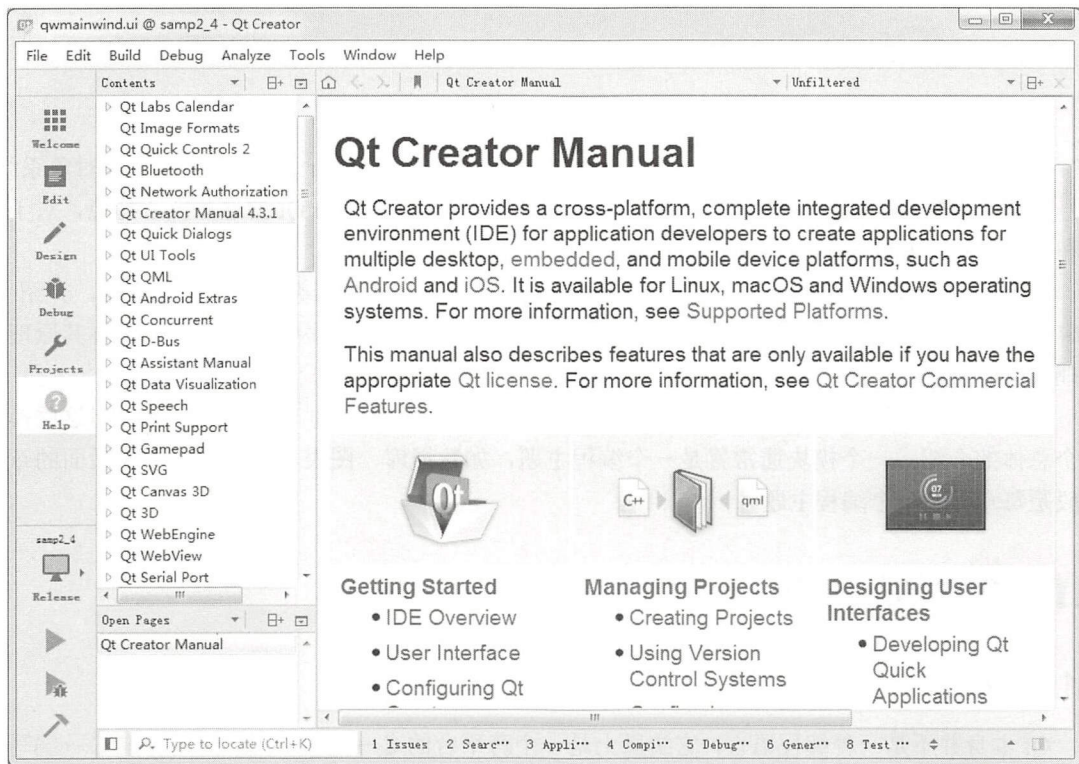


图 2-22 使用 Qt 的帮助系统查看资料

在帮助文件显示界面上，左上方工具栏中有个下拉列表框，可以选择 Bookmarks、Contents、Index 和 Search 4 种模式。

- Bookmarks 模式下，左边框里显示已存储的 Bookmarks（书签），任何帮助页面下，点击窗口上方工具栏上的“Add Bookmark”可以添加书签。
- Contents 模式下，左边框里以目录树形式显示 Qt 的所有模块（如图 2-22 所示），可以分类浏览想看的内容。
- Index 模式下，可以输入查找内容，左边框里会列出与输入内容前匹配的帮助用户列表。
- Search 模式下，可以输入关键字进行搜索。

在 Qt 帮助系统里可以搜索查看每个类的详细资料，如 QTextEdit，可以看到这个类的详细资料，包括在这个类定义的公共类型、属性、公共函数、信号、公共槽等。

另外，若要查看类的继承关系，可以访问 Qt 官网的“Inheritance Hierarchy”页面。



## 第3章

# Qt 类库概述

Qt 是一个用标准 C++ 编写的跨平台开发类库, 它对标准 C++ 进行了扩展, 引入了元对象系统、信号与槽、属性等特性, 使应用程序的开发变得更高效。本章将介绍 Qt 的这些核心特点, 对于理解和编写高效的 Qt C++ 程序是大有帮助的。

本章还介绍 `<QtGlobal>` 头文件中 Qt 的一些全局定义, 包括数据类型、函数和宏等, 介绍 Qt 的容器类及其相应迭代器的使用方法。这些全局定义和容器类在程序中经常用到, 了解其原理便于理解后面遇到的一些实例程序。

Qt 类库中大量的类是以模块形式分类组织的, 包括基本模块和扩展模块等, 本章对这些模块做个总体的介绍。一个模块通常就是一个编程主题, 如数据库、图表、网络等, 本书后面的章节一般是每章介绍一个编程主题。

## 3.1 Qt 核心特点

### 3.1.1 概述

Qt 本身并不是一种编程语言, 它实质上是一个跨平台的 C++ 开发类库, 是用标准 C++ 编写的类库, 它为开发 GUI 应用程序和非 GUI 应用程序提供了各种类。

Qt 对标准 C++ 进行了扩展, 引入了一些新的概念和功能, 例如信号与槽、对象属性等。Qt 的元对象编译器 (Meta-Object Compiler, MOC) 是一个预处理器, 在源程序被编译前先将这些 Qt 特性的程序转换为标准 C++ 兼容的形式, 然后再由标准 C++ 编译器进行编译。这就是为什么在使用信号与槽机制的类里, 必须添加一个 `Q_OBJECT` 宏的原因, 只有添加了这个宏, moc 才能对类里的信号与槽的代码进行预处理。

Qt Core 模块是 Qt 类库的核心, 所有其他模块都依赖于此模块, 如果使用 qmake 来构建项目, Qt Core 模块则是被自动加入的。

Qt 为 C++ 语言增加的特性就是在 Qt Core 模块里实现的, 这些扩展特性由 Qt 的元对象系统实现, 包括信号与槽机制、属性系统、动态类型转换等。

### 3.1.2 元对象系统

Qt 的元对象系统 (Meta-Object System) 提供了对象之间通信的信号与槽机制、运行时类型信

息和动态属性系统。

元对象系统由以下三个基础组成。

- QObject 类是所有使用元对象系统的类的基类。
- 在一个类的 private 部分声明 Q\_OBJECT 宏, 使得类可以使用元对象的特性, 如动态属性、信号与槽。
- MOC (元对象编译器) 为每个 QObject 的子类提供必要的代码来实现元对象系统的特性。

构建项目时, MOC 工具读取 C++ 源文件, 当它发现类的定义里有 Q\_OBJECT 宏时, 它就会为这个类生成另外一个包含有元对象支持代码的 C++ 源文件, 这个生成的源文件连同类的实现文件一起被编译和连接。

除了信号与槽机制外, 元对象还提供如下一些功能。

- QObject::metaObject() 函数返回类关联的元对象, 元对象类 QMetaObject 包含了访问元对象的一些接口函数, 例如 QMetaObject::className() 函数可在运行时返回类的名称字符串。

```
QObject *obj = new QPushButton;
obj->metaObject()->className(); // 返回 "QPushButton"
```

- QMetaObject::newInstance() 函数创建类的一个新的实例。
- QObject::inherits(const char \*className) 函数判断一个对象实例是否是名称为 className 的类或 QObject 的子类的实例。例如:

```
QTimer *timer = new QTimer; // QTimer 是 QObject 的子类
timer->inherits("QTimer"); // 返回 true
timer->inherits("QObject"); // 返回 true
timer->inherits("QAbstractButton"); // 返回 false, 不是 QAbstractButton 的子类
```

- QObject::tr() 和 QObject::trUtf8() 函数可翻译字符串, 用于多语言界面设计, 在第 16 章会专门介绍多语言界面设计。
- QObject::setProperty() 和 QObject::property() 函数用于通过属性名称动态设置和获取属性值。

对于 QObject 及其子类, 还可以使用 qobject\_cast() 函数进行动态投射 (dynamic cast)。例如, 假设 QMyWidget 是 QWidget 的子类并且在类定义中声明了 Q\_OBJECT 宏。创建实例使用下面的语句:

```
QObject *obj = new QMyWidget;
```

变量 obj 定义为 QObject 指针, 但它实际指向 QMyWidget 类, 所以可以正确投射为 QWidget, 即:

```
QWidget *widget = qobject_cast<QWidget *>(obj);
```

从 QObject 到 QWidget 的投射是成功的, 因为 obj 实际是 QMyWidget 类, 是 QWidget 的子类。也可以将其成功投射为 QMyWidget, 即:

```
QMyWidget *myWidget = qobject_cast<QMyWidget *>(obj);
```

投射为 QMyWidget 是成功的, 因为 qobject\_cast() 并不区分 Qt 内建的类型和用户自定义类型。

但是, 若要将 obj 投射为 QLabel 则是失败的, 即:

```
QLabel *label = qobject_cast<QLabel *>(obj);
```

这样投射是失败的，返回指针 label 为 NULL，因为 QMyWidget 不是 QLabel 的子类。

使用动态投射，使得程序可以在运行时对不同的对象做不同的处理。

### 3.1.3 属性系统

#### 1. 属性定义

Qt 提供一个 Q\_PROPERTY()宏可以定义属性，它也是基于元对象系统实现的。Qt 的属性系统与 C++编译器无关，可以用任何标准的 C++编译器编译定义了属性的 Qt C++程序。

在 QObject 的子类中，用宏 Q\_PROPERTY()定义属性，其使用格式如下：

```
Q_PROPERTY(type name
           (READ getFunction [WRITE setFunction] |
            MEMBER memberName [(READ getFunction | WRITE setFunction)])
           [RESET resetFunction]
           [NOTIFY notifySignal]
           [REVISION int]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
           [USER bool]
           [CONSTANT]
           [FINAL] )
```

Q\_PROPERTY 宏定义一个返回值类型为 type，名称为 name 的属性，用 READ、WRITE 关键字定义属性的读取、写入函数，还有其他的一些关键字定义属性的一些操作特性。属性的类型可以是 QVariant 支持的任何类型，也可以用户自定义类型。

Q\_PROPERTY 宏定义属性的一些主要关键字的意义如下。

- READ 指定一个读取属性值的函数，没有 MEMBER 关键字时必须设置 READ。
- WRITE 指定一个设定属性值的函数，只读属性没有 WRITE 设置。
- MEMBER 指定一个成员变量与属性关联，成为可读可写的属性，无需再设置 READ 和 WRITE。
- RESET 是可选的，用于指定一个设置属性缺省值的函数。
- NOTIFY 是可选的，用于设置一个信号，当属性值变化时发射此信号。
- DESIGNABLE 表示属性是否在 Qt Designer 里可见，缺省为 true。
- CONSTANT 表示属性值是一个常数，对于一个对象实例，READ 指定的函数返回值是常数，但是每个实例的返回值可以不一样。具有 CONSTANT 关键字的属性不能有 WRITE 和 NOTIFY 关键字。
- FINAL 表示所定义的属性不能被子类重载。

QWidget 类定义属性的一些例子如下：

```
Q_PROPERTY(bool focus READ hasFocus)
Q_PROPERTY(bool enabled READ isEnabled WRITE setEnabled)
Q_PROPERTY(QCursor cursor READ cursor WRITE setCursor RESET unsetCursor)
```

#### 2. 属性的使用

不管是否用 READ 和 WRITE 定义了接口函数，只要知道属性名称，就可以通过 QObject::property()



读取属性值，并通过 `QObject::setProperty()` 设置属性值。例如：

```
QPushButton *button = new QPushButton;
QObject *object = button;
object->setProperty("flat", true);
bool isFlat= object->property("flat");
```

### 3. 动态属性

`QObject::setProperty()` 函数可以在运行时为类定义一个新的属性，称之为动态属性。动态属性是针对类的实例定义的。

动态属性可以使用 `QObject::property()` 查询，就如在类定义里用 `Q_PROPERTY` 宏定义的属性一样。

例如，在数据表编辑界面上，一些字段是必填字段，就可以在初始化界面时为这些字段的关联显示组件定义一个新的 `required` 属性，并设置值为“true”，如：

```
editName->setProperty("required", "true");
comboSex-> setProperty("required", "true");
checkAgree-> setProperty("required", "true");
```

然后，可以应用下面的样式定义将这种必填字段的背景颜色设置为亮绿色（样式定义详见 16.2 节）。

```
*[required="true"] {background-color: lime}
```

### 4. 类的附加信息

属性系统还有一个宏 `Q_CLASSINFO()`，可以为类的元对象定义“名称——值”信息，如：

```
class QMyClass : public QObject
{
    Q_OBJECT
    Q_CLASSINFO("author", "Wang")
    Q_CLASSINFO("company", "UPC")
    Q_CLASSINFO("version ", "3.0.1")
public:
    ...
};
```

用 `Q_CLASSINFO()` 宏定义附加类信息后，可以通过元对象的一些函数获取类的附加信息，如 `classInfo(int)` 获取某个附加信息，函数原型定义如下：

```
QMetaClassInfo QMetaObject::classInfo(int index) const
```

返回值是 `QMetaClassInfo` 类型，有 `name()` 和 `value()` 两个函数，可获得类附加信息的名称和值。

## 3.1.4 信号与槽

在第 2 章中已经初步介绍了信号与槽的使用。信号与槽是 Qt 的一个核心特点，也是它区别于其他框架的重要特性。信号与槽是对象间进行通信的机制，也需要由 Qt 的元对象系统支持才能实现的。

Qt 使用信号与槽的机制实现对象间通信，它隐藏了复杂的底层实现，完成信号与槽的关联后，发射信号时并不需要知道 Qt 是如何找到槽函数的。Qt 的信号与槽机制与 Delphi 和 C++ Builder 的“事件——响应”比较类似，但是更加灵活。

某些开发架构使用回调函数（callback）实现对象间通信。与回调函数相比，信号与槽的执行速度稍微慢一点，因为需要查找连接的对象和槽函数，但是这种差别在应用程序运行时是感觉不到的，而其提供的灵活性却比回调函数强很多。

信号与槽的基本特点和使用方法在 2.2 节已有介绍，此处对信号与槽的特点和用法做一些补充。

### 1. connect()函数的不同参数形式

QObject::connect()函数有多重参数形式，一种参数形式的函数原型是：

```
QMetaObject::Connection QObject::connect(const QObject *sender, const char *signal,
const QObject *receiver, const char *method, Qt::ConnectionType type = Qt::AutoConnection)
```

使用这种参数形式的 connect()进行信号与槽函数的连接时，一般句法如下：

```
connect(sender, SIGNAL(signal()), receiver, SLOT(slot()));
```

这里使用了宏 SIGNAL()和 SLOT()指定信号和槽函数，而且如果信号和槽函数带有参数，还需注明参数类型，如：

```
connect(spinNum, SIGNAL(valueChanged (int)), this, SLOT(updateStatus(int));
```

另外一种参数形式的 connect()函数的原型是：

```
QMetaObject::Connection QObject::connect(const QObject *sender, const QMetaMethod
&signal, const QObject *receiver, const QMetaMethod &method, Qt::ConnectionType type =
Qt::AutoConnection)
```

对于具有默认参数的信号与槽（即信号名称是唯一的，没有参数不同而同名的两个信号），可以使用这种函数指针形式进行关联，如：

```
connect(lineEdit, &QLineEdit::textChanged, this, &widget::on_textChanged);
```

QLineEdit 只有一个信号 textChanged(QString)，在自定义窗体类 widget 里定义一个槽函数 on\_textChanged(QString)，就可以用上面的语句将此信号与槽关联起来，无需出现函数参数。这在信号的参数比较多时更简便一些。

而对于具有不同参数的同名信号就不能采用函数指针的方式进行信号与槽的关联，例如 QSpinBox 有两个 valueChanged()信号，分别是：

```
void QSpinBox::valueChanged(int i)
void QSpinBox::valueChanged(const QString &text)
```

即使在自定义窗体 widget 里定义了一个槽函数，如：

```
void onValueChanged(int i);
```

在使用下面的语句进行关联时，编译会出错。

```
connect(spinNum, &QSpinBox::valueChanged, this, &widget::onValueChanged);
```

不管是哪种参数形式的 connect()函数，最后都有一个参数 Qt::ConnectionType type，缺省值为 Qt::AutoConnection。枚举类型 Qt::ConnectionType 表示了信号与槽之间的关联方式，有以下几种取值。

- Qt::AutoConnection(缺省值)：如果信号的接收者与发射者在同一个线程，就使用 Qt::Direct Connection 方式；否则使用 Qt::QueuedConnection 方式，在信号发射时自动确定关联方式。

- Qt::DirectConnection: 信号被发射时槽函数立即执行, 槽函数与信号在同一个线程。
- Qt::QueuedConnection: 在事件循环回到接收者线程后执行槽函数, 槽函数与信号在不同的线程。
- Qt::BlockingQueuedConnection: 与 Qt::QueuedConnection 相似, 只是信号线程会阻塞直到槽函数执行完毕。当信号与槽函数在同一个线程时绝对不能使用这种方式, 否则会造成死锁。

## 2. 使用 sender() 获得信号发射者

在槽函数里, 使用 QObject::sender() 可以获取信号发射者的指针。如果知道信号发射者的类型, 可以将指针投射为确定的类型, 然后使用这个确定类的接口函数。

例如, 在 QSpinBox 的 valueChanged(int) 信号的槽函数里, 可以通过 sender() 和 qobject\_cast 获得信号发射者的指针, 从而对信号发射者进行操作。

```
QSpinBox *spinBox = qobject_cast<QSpinBox *>(sender());
```

## 3. 自定义信号及其使用

在自己设计的类里也可以自定义信号, 信号就是在类定义里声明的一个函数, 但是这个函数无需实现, 只需发射 (emit)。

例如, 在下面的自定义类 QPerson 的 signals 部分定义一个信号 ageChanged(int)。

```
class QPerson : public QObject
{
    Q_OBJECT
private:
    int m_age=10;
public:
    void incAge();
signals:
    void ageChanged( int value);
}
```

信号函数必须是无返回值的函数, 但是可以有输入参数。信号函数无需实现, 只需在某些条件下发射信号。例如, 在 incAge() 函数中发射信号, 其代码如下。

```
void QPerson::incAge()
{
    m_age++;
    emit ageChanged(m_age); //发射信号
}
```

在 incAge() 函数里, 当私有变量 m\_age 变化后, 发射信号 ageChanged(int), 表示年龄发生了变化。至于是否有与此信号相关联的槽函数, 信号发射者并不管。如果在使用 QPerson 类对象的程序中为此信号关联了槽函数, 在 incAge() 函数里发射此信号时, 就会执行相关联的槽函数。至于是否立即执行槽函数, 发射信号的线程是否等待槽函数执行完之后再执行后面的代码, 与 connect() 函数设置信号与槽关联时设置的连接类型以及信号与槽是否在同一个线程有关。

### 3.1.5 元对象特性测试实例

#### 1. QPerson 类的定义

实例 samp3\_1 演示 Qt 元对象系统的一些功能。samp3\_1 是主窗口基于 QWidget 的应用程序,



在项目创建后, 创建一个类 `QPerson`。 `qperson.h` 文件中的类定义如下所示:

```
class QPerson : public QObject
{
    Q_OBJECT
    Q_CLASSINFO("author", "Wang")
    Q_CLASSINFO("company", "UPC")
    Q_CLASSINFO("version", "1.0.0")
    Q_PROPERTY(int age READ age WRITE setAge NOTIFY ageChanged)
    Q_PROPERTY(QString name MEMBER m_name)
    Q_PROPERTY(int score MEMBER m_score)
private:
    int m_age=10;
    QString m_name;
    int m_score=79;
public:
    explicit QPerson(QString fName, QObject *parent = nullptr);
    int age();
    void setAge(int value);
    void incAge();
signals:
    ageChanged( int value);
public slots:
};
```

`QPerson` 是 `QObject` 的子类, 在类定义部分使用了宏 `Q_OBJECT`, 这样 `QPerson` 就获得了元对象系统支持, 能使用信号与槽、属性等功能。

`QPerson` 使用 `Q_CLASSINFO` 宏定义了 3 个附加信息, 使用 `Q_PROPERTY` 宏定义了 3 个属性, 还定义了一个信号。下面是 `QPerson` 类的实现代码:

```
QPerson::QPerson(QString fName, QObject *parent) : QObject(parent)
{ //构造函数
    m_name=fName;
}
int QPerson::age() //返回 age
{ return m_age;
}
void QPerson::setAge(int value) //设置 age
{ m_age=value;
    emit ageChanged(m_age); //发射信号
}
void QPerson::incAge()
{ m_age++;
    emit ageChanged(m_age); //发射信号
}
```

`setAge(int)` 函数用于设置年龄, 代码里设置年龄后发射信号 `ageChanged()`。

`incAge()` 是一个单独的接口函数, 与属性无关, 但是也发射信号 `ageChanged()`。

## 2. 元对象特性的使用

实例 `samp3_1` 运行时界面如图 3-1 所示。

主窗口是基于 `QWidget` 的可视化设计的类 `QmyWidget`, 其定义如下:



图 3-1 实例 `samp3_1` 运行时界面

```

class QmyWidget : public QWidget
{
    Q_OBJECT
private:
    QPerson *boy;
    QPerson *girl;
public:
    explicit QmyWidget(QWidget *parent = 0);
    ~QmyWidget();
private:
    Ui::QmyWidget *ui;
private slots:
    //自定义槽函数
    void on_ageChanged(int value);
    void on_spin_valueChanged(int arg1);
    //界面按钮的槽函数
    void on_btnClear_clicked();
    void on_btnBoyInc_clicked();
    void on_btnGirlInc_clicked();
    void on_btnClassInfo_clicked();
}

```

`QmyWidget` 类里定义了两个 `QPerson` 类型的指针变量，定义了两个自定义槽函数。构造函数的代码如下：

```

QmyWidget::QmyWidget(QWidget *parent) : QWidget(parent), ui(new Ui::QmyWidget)
{
    //构造函数
    ui->setupUi(this);
    boy=new QPerson("王小明");
    boy->setProperty("score",95);
    boy->setProperty("age",10);
    boy->setProperty("sex","Boy");//动态属性
    connect(boy,&QPerson::ageChanged,this,&QmyWidget::on_ageChanged);

    girl=new QPerson("张小丽");
    girl->setProperty("score",81);
    girl->setProperty("age",20);
    girl->setProperty("sex","Girl");//动态属性
    connect(girl,&QPerson::ageChanged,this,&QmyWidget::on_ageChanged);
    ui->spinBoy->setProperty("isBoy",true); //动态属性
    ui->spinGirl->setProperty("isBoy",false);

    connect(ui->spinGirl,SIGNAL(valueChanged(int)),
            this,SLOT(on_spin_valueChanged(int)));
    connect(ui->spinBoy,SIGNAL(valueChanged(int)),
            this,SLOT(on_spin_valueChanged(int)));
}

```

创建 `QPerson` 类型对象 `boy` 后，使用 `setProperty()` 函数设置了 `score`、`age` 属性的值，这两个属性是 `QPerson` 类里定义的。还设置了一个属性 `sex` 的值。

```
boy->setProperty("sex","Boy");
```

`sex` 属性在 `QPerson` 类里没有定义，所以这个属性是个动态属性。

创建对象 `boy` 和 `girl` 后，它们的 `ageChanged()` 信号都与槽函数 `on_ageChanged()` 关联，设置信

号与槽关联时使用了函数指针的形式, 因为 QPerson 只有一个 ageChanged() 信号, 具有缺省函数参数, 这样设置关联是可以的。

为界面上的组件 spinBoy 和 spinGirl 也设置了一个逻辑型动态属性 isBoy, 分别赋值为 true 和 false, 并且这两个 spinBox 的信号 valueChanged(int) 都与槽函数 on\_spin\_valueChanged(int) 相关联。但是, 这里使用 connect() 函数设置关联时必须使用 SIGNAL 和 SLOT 宏的形式, 而不能使用函数指针的形式, 因为 QSpinBox 有两个 valueChanged() 信号, 只是参数不同。

自定义槽函数 on\_ageChanged() 用于响应 QPerson 的 ageChanged() 信号, 其实现代码如下:

```
void QmyWidget::on_ageChanged( int value)
{ // 响应 QPerson 的 ageChanged() 信号
    Q_UNUSED(value);
    QPerson *aPerson = qobject_cast<QPerson *>(sender()); // 类型投射
    QString hisName=aPerson->property("name").toString(); // 姓名
    QString hisSex=aPerson->property("sex").toString(); // 动态属性
    int hisAge=aPerson->age(); // 通过接口函数获取年龄
    // int hisAge=aPerson->property("age").toInt(); // 通过属性获得年龄
    ui->textEdit->appendPlainText(hisName+" "+hisSex
                                +QString::asprintf(", 年龄=%d", hisAge));
}
```

这里使用了 QObject::sender() 函数获取信号发射者。因为信号发射者是 QPerson 类型对象 boy 或 girl, 所以可以用 qobject\_cast() 将发射者投射为具体的类型。

```
QPerson *aPerson = qobject_cast<QPerson *>(sender());
```

这样得到信号发射者 QPerson 类型的对象指针 aPerson, 它指向 boy 或 girl。

使用 aPerson 指针, 通过 property() 函数获取 name 属性的值, 也可以获取动态属性 sex 的值。因为在 QPerson 中, name 属性只用 MEMBER 关键字定义了一个私有变量表示这个属性, 所以只能用 property() 读取此属性的值, 也只能用 setProperty() 设置此属性的值。

读取年龄时, 直接用了接口函数, 即:

```
int hisAge=aPerson->age();
```

当然也可以采用 property() 函数获取年龄, 即:

```
int hisAge=aPerson->property("age").toInt();
```

因为定义 age 属性时用 READ 和 WRITE 指定了公共的接口函数, 既可以通过 property() 和 setProperty() 进行属性读写, 也可以直接使用接口函数进行读写。当然, 直接使用接口函数速度更快。

界面上两个分别用于设置 boy 和 girl 年龄的 spinBox 的 valueChanged(int) 信号与槽函数 on\_spin\_valueChanged(int) 关联, 槽函数代码如下:

```
void QmyWidget::on_spin_valueChanged(int arg1)
{ // 响应界面上 spinBox 的 valueChanged(int) 信号
    Q_UNUSED(arg1);
    QSpinBox *spinBox = qobject_cast<QSpinBox *>(sender());
    if (spinBox->property("isBoy").toBool())
        boy->setAge(spinBox->value());
    else
```



```
girl->setAge(spinBox->value());
}
```

这里也使用了信号发射者的类型投射, 投射为 `QSpinBox` 类型指针 `spinBox`, 然后根据 `spinBox` 的动态属性 `isBoy` 的值, 确定调用 `boy` 或 `girl` 的 `setAge()` 函数。

这种编写代码的方式一般用于为多个同类型组件的同一信号编写一个槽函数, 在槽函数里区分信号来源分别做处理, 避免为每个组件分别编写槽函数形成的代码冗余。

界面上“类的元对象信息”按钮的响应代码如下:

```
void QmyWidget::on_btnClassInfo_clicked()
{
    // "类的元对象信息" 按钮
    const QMetaObject *meta = boy->metaObject();
    ui->textEdit->clear();
    ui->textEdit->appendPlainText("==元对象信息==\n");
    ui->textEdit->appendPlainText(
        QString("类名称: %1\n").arg(meta->className()));
    ui->textEdit->appendPlainText("property");
    for (int i = meta->propertyOffset(); i < meta->propertyCount(); i++)
    {
        QMetaProperty prop = meta->property(i);
        const char* propName = prop.name();
        QString propValue = boy->property(propName).toString();
        ui->textEdit->appendPlainText(
            QString("属性名称=%1, 属性值=%2").arg(propName).arg(propValue));
    }

    ui->textEdit->appendPlainText("");
    ui->textEdit->appendPlainText("classInfo");
    for (int i = meta->classInfoOffset(); i < meta->classInfoCount(); i++)
    {
        QMetaClassInfo classInfo = meta->classInfo(i);
        ui->textEdit->appendPlainText(QString("Name=%1; Value=%2")
            .arg(classInfo.name()).arg(classInfo.value()));
    }
}
```

代码里通过 `boy->metaObject()` 获得对象 `boy` 的元对象。元对象类 `QMetaObject` 封装了访问类的元对象的各种接口函数, 例如, `QMetaObject::className()` 返回类的名称。

`QMetaObject` 用于属性操作的函数有以下几种。

- `propertyOffset()`: 返回类的第一个属性的序号, 第一个属性的序号不一定是 0。
- `propertyCount()`: 返回类的属性个数。
- `QMetaProperty property(int index)`: 返回序号为 `index` 的属性对象, 返回值是 `QMetaProperty` 类型, 它封装了对属性的更多特征查询功能, 以及属性值的读写功能。

`QMetaClassInfo` 类封装了类的 `classInfo` 的访问接口函数, 只有 `name()` 和 `value()` 两个接口函数。

## 3.2 Qt 全局定义

<QtGlobal> 头文件包含了 Qt 类库的一些全局定义, 包括基本数据类型、函数和宏, 一般的

Qt 类的头文件都会包含该文件，所以不用显式包含这个头文件也可以使用其中的定义。

3.2.1 数据类型定义

为了确保在各个平台上各数据类型都有统一确定的长度，Qt 为各种常见数据类型定义了类型符号，如 qint8 就是 signed char 的类型定义，即：

```
typedef signed char qint8;
```

<QtGlobal>中定义的数据类型见表 3-1。

表 3-1 Qt 中的数据类型定义

Qt 数据类型	等效定义	字节数
qint8	signed char	1
qint16	signed short	2
qint32	signed int	4
qint64	long long int	8
qlonglong	long long int	8
quint8	unsigned char	1
quint16	unsigned short	2
quint32	unsigned int	4
quint64	unsigned long long int	8
qlonglong	unsigned long long int	8
uchar	unsigned char	1
ushort	unsigned short	2
uint	unsigned int	4
ulong	unsigned long	8
qreal	double	8
qfloat16		2

其中 qreal 缺省是 8 字节 double 类型浮点数，如果 Qt 使用 -qreal float 选项进行配置，就是 4 字节 float 类型的浮点数。

qfloat16 是 Qt 5.9.0 中新增的一个类，用于表示 16 位的浮点数，要使用 qfloat16，需要包含头文件<QFloat16>。

3.2.2 函数

<QtGlobal>头文件包含一些常用函数的定义，这些函数多以模板类型作为参数，返回相应的模板类型，模板类型可以用任何其他类型替换。若是以 double 或 float 类型数作为参数的，一般有两个参数版本的同名函数，如 qFuzzyIsNull(double d) 和 qFuzzyIsNull(float f)。

表 3-2 是<QtGlobal>中常用的全局函数定义，列出了函数的输入和输出参数（若存在 double 和 float 两种参数版本，只列出 double 类型参数的版本）。

表 3-2 <QtGlobal>中常用的全局函数定义

函数	功能
T qAbs(const T &value)	返回变量 value 的绝对值
const T &qBound(const T &min, const T &value, const T &max)	返回 value 限定在 min 至 max 范围内的值
bool qFuzzyCompare(double p1, double p2)	若 p1 和 p2 近似相等，返回 true
bool qFuzzyIsNull(double d)	如果参数 d 约等于 0，返回 true

续表

函数	功能
double qInf()	返回无穷大的数
bool qIsFinite(double d)	若 d 是一个有限的数, 返回 true
bool qIsInf(double d)	若 d 是一个无限大的数, 返回 true
bool qIsNaN(double d)	若 d 不是一个数, 返回 true
const T &qMax(const T &value1, const T &value2)	返回 value1 和 value2 中较大的值
const T &qMin(const T &value1, const T &value2)	返回 value1 和 value2 中较小的值
qint64 qRound64(double value)	将 value 近似为最接近的 qint64 整数
int qRound(double value)	将 value 近似为最接近的 int 整数
int qrand()	标准 C++中 rand()函数的线程安全型版本, 返回 0 至 RAND_MAX 之间的伪随机数
void qsrand(uint seed)	标准 C++中 srand()函数的线程安全型版本, 使用种子 seed 对伪随机数序列初始化

还有一些基础的数学运算函数在<QtMath>头文件中定义, 比如三角运算函数、弧度与角度之间的转换函数等。

3.2.3 宏定义

<QtGlobal>头文件中定义了很多宏, 以下一些是比较常用的。

- QT\_VERSION

这个宏展开为数值形式 0xMMNNPP (MM = major, NN = minor, PP = patch)表示 Qt 编译器版本, 例如 Qt 编译器版本为 Qt 5.9.1, 则 QT\_VERSION 为 0x050901。这个宏常用于条件编译设置, 根据 Qt 版本不同, 编译不同的代码段。

```
#if QT_VERSION >= 0x040100
    QIcon icon = style()->standardIcon(QStyle::SP_TrashIcon);
#else
    QPixmap pixmap = style()->standardPixmap(QStyle::SP_TrashIcon);
    QIcon icon(pixmap);
#endif
```

- QT\_VERSION\_CHECK

这个宏展开为 Qt 版本号的一个整数表示, 例如:

```
#if (QT_VERSION >= QT_VERSION_CHECK(5, 0, 0))
#include <QtWidgets>
#else
#include <QtGui>
#endif
```

- QT\_VERSION\_STR

这个宏展开为 Qt 版本号的字符串, 如 “5.9.0”。

- Q\_BYTE\_ORDER、Q\_BIG\_ENDIAN 和 Q\_LITTLE\_ENDIAN

Q\_BYTE\_ORDER 表示系统内存中数据的字节序, Q\_BIG\_ENDIAN 表示大端字节序, Q\_LITTLE\_ENDIAN 表示小端字节序。在需要判断系统字节序时会用到, 例如:

```
#if Q_BYTE_ORDER == Q_LITTLE_ENDIAN
...
...
...
#endif
```



```
#endif
```

- `Q_DECL_IMPORT` 和 `Q_DECL_EXPORT`

在使用或设计共享库时，用于导入或导出库的内容，12.4 节有其使用实例。

- `Q_DECL_OVERRIDE`

在类定义中，用于重载一个虚函数，例如在某个类中重载虚函数 `paintEvent()`，可以定义如下：

```
void paintEvent(QPaintEvent*) Q_DECL_OVERRIDE;
```

使用 `Q_DECL_OVERRIDE` 宏后，如果重载的虚函数没有进行任何重载操作，编译器将会报错。

- `Q_DECL_FINAL`

这个宏将一个虚函数定义为最终级别，不能再被重载，或定义一个类不能再被继承，示例如下：

```
class QRect Q_DECL_FINAL { // QRect 不能再被继承
    // ...
};
```

- `Q_UNUSED(name)`

这个宏用于在函数中定义不在函数体里使用的参数，示例如下：

```
void MainWindow::on_imageSaved(int id, const QString &fileName)
{
    Q_UNUSED(id);
    LabInfo->setText("图片保存为: "+fileName);
}
```

在这个函数里，`id` 参数没有使用。如果不用 `Q_UNUSED(id)` 定义，编译器会出现参数未使用的警告。

- `foreach(variable, container)`

`foreach` 用于容器类的遍历，例如：

```
foreach (const QString &codecName, recorder->supportedAudioCodecs())
    ui->comboBox->addItem(codecName);
```

- `forever`

`forever` 用于构造一个无限循环，例如：

```
forever {
    ...
}
```

- `qDebug(const char *message, ...)`

在 debugger 窗体显示信息，如果编译器设置了 `Qt_NO_DEBUG_OUTPUT`，则不作任何输出，例如：

```
qDebug("Items in list: %d", myList.size());
```

类似的宏还有 `qWarning`、`qCritical`、`qFatal`、`qInfo` 等，也是用于在 debugger 窗体显示信息。

## 3.3 容器类

### 3.3.1 容器类概述

Qt 提供了多个基于模板的容器类，这些容器类可以用于存储指定类型的数据项，例如常用的

字符串列表类 `QStringList` 就是从容器类 `QList<QString>` 继承的，实现对字符串列表的添加、存储、删除等操作。

Qt 的容器类比标准模板库（STL）中的容器类更轻巧、安全和易于使用。这些容器类是隐式共享和可重入的，而且它们进行了速度和存储优化，因此可以减少可执行文件的大小。此外，它们还是线程安全的，也就是说它们作为只读容器时可被多个线程访问。

容器类是基于模板的类，如常用的容器类 `QList<T>`，`T` 是一个具体的类型，可以是 `int`、`float` 等简单类型，也可以是 `QString`、`QDate` 等类，但不能是 `QObject` 或任何其子类。`T` 必须是一个可赋值的类型，即 `T` 必须提供一个缺省的构造函数，一个可复制构造函数和一个赋值运算符。

例如用 `QList<T>` 定义一个字符串列表的容器，其定义方法是：

```
QList<QString> aList;
```

这样定义了一个 `QList` 容器类的变量 `aList`，它的数据项是 `QString`，所以 `aList` 可以用于处理字符串列表，例如：

```
aList.append("Monday");
aList.append("Tuesday");
aList.append("Wednesday");
QString str=aList[0];
```

Qt 的容器类分为顺序容器（sequential containers）和关联容器（associative containers）。

容器迭代类用于遍历容器里的数据项，有 Java 类型的迭代类和 STL 类型的迭代类。Java 类型的迭代类易于使用，提供高级功能，而 STL 类型的迭代类效率更高一些。

Qt 还提供了 `foreach` 宏用于遍历容器内的所有数据项。

### 3.3.2 顺序容器类

Qt 的顺序容器类有 `QList`、`QLinkedList`、`QVector`、`QStack` 和 `QQueue`。

#### 1. QList

`QList` 是最常用的容器类，虽然它是以数组列表（array-list）的形式实现的，但是在其前或后添加数据非常快，`QList` 以下标索引的方式对数据项进行访问。

`QList` 用于添加、插入、替换、移动、删除数据项的函数有：`insert()`、`replace()`、`removeAt()`、`move()`、`swap()`、`append()`、`prepend()`、`removeFirst()`和 `removeLast()`等。

`QList` 提供下标索引方式访问数据项，如同数组一样，也提供 `at()`函数，例如：

```
QList<QString> list;
list << "one" << "two" << "three";
QString str1=list[1];    //str1=="two"
QString str0=list.at(0); //str0=="one"
```

`QList` 的 `isEmpty()`函数在数据项为空时返回 `true`，`size()`函数返回数据项的个数。

`QList` 是 Qt 中最常用的容器类，很多函数的参数传递都是采用 `QList` 容器类，例如 `QAudioDeviceInfo` 的静态函数 `availableDevices()`的函数原型是：

```
QList<QAudioDeviceInfo> QAudioDeviceInfo::availableDevices(QAudio::Mode mode)
```

其返回数据就是 `QAudioDeviceInfo` 类型的 `QList` 列表。

## 2. `QLinkedList`

`QLinkedList<T>` 是链式列表 (linked-list)，数据项不是用连续的内存存储的，它基于迭代器访问数据项，并且插入和删除数据项的操作时间相同。

除了不提供基于下标索引的数据项访问外，`QLinkedList` 的其他接口函数与 `QList` 基本相同。

## 3. `QVector`

`QVector<T>` 提供动态数组的功能，以下标索引访问数据。

`QVector` 的函数接口与 `QList` 几乎完全相同，`QVector<T>` 的性能比 `QList<T>` 更高，因为 `QVector<T>` 的数据项是连续存储的。

## 4. `QStack`

`QStack<T>` 是提供类似于堆栈的后入先出 (LIFO) 操作的容器类，`push()` 和 `pop()` 是主要的接口函数。例如：

```
QStack<int> stack;
stack.push(10);
stack.push(20);
stack.push(30);
while (!stack.isEmpty())
    cout << stack.pop() << endl;
```

程序会依次输出 30, 20, 10。

## 5. `QQueue`

`QQueue<T>` 是提供类似于队列先入先出 (FIFO) 操作的容器类。`enqueue()` 和 `dequeue()` 是主要操作函数。例如：

```
QQueue<int> queue;
queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);
while (!queue.isEmpty())
    cout << queue.dequeue() << endl;
```

程序会依次输出 10, 20, 30。

### 3.3.3 关联容器类

Qt 还提供关联容器类 `QMap`、`QMultiMap`、`QHash`、`QMultiHash` 和 `QSet`。

`QMultiMap` 和 `QMultiHash` 支持一个键关联多个值，`QHash` 和 `QMultiHash` 类使用散列 (Hash) 函数进行查找，查找速度更快。

#### 1. `QSet`

`QSet` 是基于散列表的集合模板类，它存储数据的顺序是不定的，查找值的速度非常快。

`QSet<T>` 内部就是用 `QHash` 实现的。

定义 `QSet<T>` 容器和输入数据的实例代码如下：

```
QSet<QString> set;
```



```
set << "dog" << "cat" << "tiger";
```

测试一个值是否包含于这个集合，用 `contains()` 函数，示例如下：

```
if (!set.contains("cat"))
    ...
```

## 2. QMap

`QMap<Key, T>` 提供一个字典（关联数组），一个键映射到一个值。`QMap` 存储数据是按照键的顺序，如果不在乎存储顺序，使用 `QHash` 会更快。

定义 `QMap<QString, int>` 类型变量和赋值的示例代码如下：

```
QMap<QString, int> map;
map["one"] = 1;
map["two"] = 2;
map["three "] = 3;
```

也可以使用 `insert()` 函数赋值，或 `remove()` 移除一个键值对，示例如下：

```
map.insert("four", 4);
map.remove("two");
```

要查找一个值，使用运算符 “[ ]” 或 `value()` 函数，示例如下：

```
int num1 = map["one"];
int num2 = map.value("two");
```

如果在映射表中没有找到指定的键，会返回一个缺省构造值（default-constructed values），例如，如果值的类型是字符串，会返回一个空的字符串。

在使用 `value()` 函数查找键值时，还可以指定一个缺省的返回值，示例如下：

```
timeout = map.value("TIMEOUT", 30);
```

这表示如果在 `map` 里找到键 “TIMEOUT”，就返回关联的值，否则返回值为 30。

## 3. QMapMultiMap

`QMultiMap` 是 `QMap` 的子类，是用于处理多值映射的便利类。

多值映射就是一个键可以对应多个值。`QMap` 正常情况下不允许多值映射，除非使用 `QMap::insertMulti()` 添加键值对。

`QMultiMap` 是 `QMap` 的子类，所以 `QMap` 的大多数函数在 `QMultiMap` 都是可用的，但是有几个特殊的，`QMultiMap::insert()` 等效于 `QMap::insertMulti()`，`QMultiMap::replace()` 等效于 `QMap::insert()`。

`QMultiMap` 使用示例如下：

```
QMultiMap<QString, int> map1, map2, map3;
map1.insert("plenty", 100);
map1.insert("plenty", 2000); // map1.size() == 2
map2.insert("plenty", 5000); // map2.size() == 1
map3 = map1 + map2;          // map3.size() == 3
```

`QMultiMap` 不提供 “[ ]” 操作符，使用 `value()` 函数访问最新插入的键的单个值。如果要获取一个键对应的所有值，使用 `values()` 函数，返回值是 `QList<T>` 类型。

```
QList<int> values = map.values("plenty");
```

```
for (int i = 0; i < values.size(); ++i)
    cout << values.at(i) << endl;
```

4. QHash

QHash 是基于散列表来实现字典功能的模板类，QHash<Key, T>存储的键值对具有非常快的查找速度。

QHash 与 QMap 的功能和用法相似，区别在于以下几点：

- QHash 比 QMap 的查找速度快；
- 在 QMap 上遍历时，数据项是按照键排序的，而 QHash 的数据项是任意顺序的；
- QMap 的键必须提供“<”运算符，QHash 的键必须提供“=”运算符和一个名称为 qHash() 的全局散列函数。

5. QMultiHash

QMultiHash 是 QHash 的子类，是用于处理多值映射的便利类，其用法与 QMultiMap 类似。

3.4 容器类的迭代

迭代器（iterator）为访问容器类里的数据项提供了统一的方法，Qt 有两种迭代器类：Java 类型的迭代器和 STL 类型的迭代器。

Java 类型的迭代器更易于使用，且提供一些高级功能，而 STL 类型的迭代器效率更高。

Qt 还提供一个关键字 foreach（实际是< QtGlobal >里定义的一个宏）用于方便地访问容器里所有数据项。

3.4.1 Java 类型迭代器

1. Java 类型迭代器总表

对于每个容器类，有两个 Java 类型迭代器：一个用于只读操作，一个用于读写操作，各个 Java 类型的容器类见表 3-3。

表 3-3 Java 类型的迭代器类

容器类	只读迭代器	读写迭代器
QList<T>, QQueue<T>	QListIterator<T>	QMutableListIterator<T>
QLinkedList<T>	QLinkedListIterator<T>	QMutableLinkedListIterator<T>
QVector<T>, QStack<T>	QVectorIterator<T>	QMutableVectorIterator<T>
QSet<T>	QSetIterator<T>	QMutableSetIterator<T>
QMap<Key, T>, QMultiMap<Key, T>	QMapIterator<Key, T>	QMutableMapIterator<Key, T>
QHash<Key, T>, QMultiHash<Key, T>	QHashIterator<Key, T>	QMutableHashIterator<Key, T>

QMap 和 QHash 等关联容器类的迭代器用法相同，QList 和 QLinkedList、QSet 等容器类的用法相同，所以下面只以 QMap 和 QList 为例介绍迭代器的用法。

2. 顺序容器类的迭代器的使用

Java 类型迭代器的指针不是指向一个数据项，而是在数据项之间，迭代器指针位置示意图如图 3-2 所示。

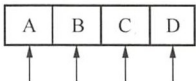


图 3-2 Java 类型迭代器位置示意图

下面是遍历访问一个 `QList<QString>` 容器的所有数据项的典型代码。

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QListIterator<QString> i(list);
while (i.hasNext())
    qDebug() << i.next();
```

`QList< QString>` 容器对象 `list` 作为参数传递给 `QListIterator< QString >` 迭代器 `i` 的构造函数，`i` 用于对 `list` 作只读遍历。起始时刻，迭代器指针在容器第一个数据项的前面（图 3-2 中数据项“A”的前面），调用 `hasNext()` 判断在迭代器指针后面是否还有数据项，如果有，就调用 `next()` 跳过一个数据项，并且 `next()` 函数返回跳过去的那个数据项。

也可以反向遍历，示例代码如下：

```
QListIterator<QString> i(list);
i.toBack();
while (i.hasPrevious())
    qDebug() << i.previous();
```

`QListIterator` 用于移动指针和读取数据的函数见表 3-4。

表 3-4 QListIterator 常用函数

函数名	功能
<code>void toFront()</code>	迭代器移动到列表的最前面（第一个数据项之前）
<code>void toBack()</code>	迭代器移动到列表的最后面（最后一个数据项之后）
<code>bool hasNext()</code>	如果迭代器不是位于列表最后位置，返回 <code>true</code>
<code>const T &amp; next()</code>	返回下一个数据项，并且迭代器后移一个位置
<code>const T &amp; peekNext()</code>	返回下一个数据项，但是不移动迭代器位置
<code>bool hasPrevious()</code>	如果迭代器不是位于列表的最前面，返回 <code>true</code>
<code>const T &amp; previous()</code>	返回前一个数据项，并且迭代器前移一个位置
<code>const T &amp; peekPrevious()</code>	返回前一个数据项，但是不移动迭代器指针

`QListIterator` 是只读访问容器内数据项的迭代器，若要在遍历过程中对容器的数据进行修改，需要使用 `QMutableListIterator`。例如下面的示例代码为删除容器中数据为奇数的项。

```
QList<int> list;
list <<1 <<2<<3<<4<<5;
QMutableListIterator<int> i(list);
while (i.hasNext()) {
    if (i.next() % 2 != 0)
        i.remove();
}
```

`remove()` 函数移除 `next()` 函数刚刚跳过的一个数据项，不会使迭代器失效。

`setValue()` 函数可以修改刚刚跳过去的的数据项的值。

### 3. 关联容器类的迭代器的使用

对于关联容器类 `QMap<Key T>`，使用 `QMapIterator` 和 `QMutableMapIterator` 迭代器类，它们具有表 3-4 所示的所有函数，主要是增加了 `key()` 和 `value()` 函数用于获取刚刚跳过的数据项的键和值。

例如，下面的代码将删除键（城市名称）里以“City”结尾的数据项。

```
QMap<QString, QString> map;
```



```
map.insert("Paris", "France");
map.insert("New York", "USA");
map.insert("Mexico City", "USA");
map.insert("Moscow", "Russia");
...
QMutableMapIterator<QString, QString> i(map);
while (i.hasNext()) {
    if (i.next().key().endsWith("City"))
        i.remove();
}
```

如果是在多值容器里遍历，可以用 `findNext()`或 `findPrevious()`查找下一个或上一个值，如下面的代码将删除上一示例代码中 `map` 里值为“USA”的所有数据项。

```
QMutableMapIterator<QString, QString> i(map);
while (i.findNext("USA"))
    i.remove();
```

3.4.2 STL 类型迭代器

1. STL 类型迭代器总表

STL 迭代器与 Qt 和 STL 的原生算法兼容，并且进行了速度优化。具体类型见表 3-5。

对于每一个容器类，都有两个 STL 类型迭代器：一个用于只读访问，一个用于读写访问。无需修改数据时一定使用只读迭代器，因为它们速度更快。

表 3-5 STL 类型的迭代器类

容器类	只读迭代器	读写迭代器
QList<T>, QQueue<T>	QList<T>::const_iterator	QList<T>::iterator
QLinkedList<T>	QLinkedList<T>::const_iterator	QLinkedList<T>::iterator
QVector<T>, QStack<T>	QVector<T>::const_iterator	QVector<T>::iterator
QSet<T>	QSet<T>::const_iterator	QSet<T>::iterator
QMap<Key, T>	QMap<Key, T>::const_iterator	QMap<Key, T>::iterator
QMultiMap<Key, T>	QMultiMap<Key, T>::const_iterator	QMultiMap<Key, T>::iterator
QHash<Key, T>	QHash<Key, T>::const_iterator	QHash<Key, T>::iterator
QMultiHash<Key, T>	QMultiHash<Key, T>::const_iterator	QMultiHash<Key, T>::iterator

注意 在定义只读迭代器和读写迭代器时的区别，它们使用了不同的关键字，`const_iterator` 定义只读迭代器，`iterator` 定义读写迭代器。此外，还可以使用 `const_reverse_iterator` 和 `reverse_iterator` 定义相应的反向迭代器。

STL 类型的迭代器是数组的指针，所以“++”运算符使迭代器指向下一个数据项，“\*”运算符返回数据项内容。与 Java 类型的迭代器不同，STL 迭代器直接指向数据项，STL 迭代器指向位置示意图如图 3-3 所示。

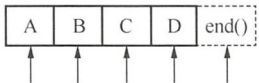


图 3-3 STL 类型迭代器位置示意图

`begin()`函数使迭代器指向容器的第一个数据项，`end()`函数使迭代器指向一个虚拟的表示结尾的数据项，`end()`表示的数据项是无效的，一般用作循环结束条件。

下面仍然以 `QList` 和 `QMap` 为例说明 STL 迭代器的用法，其他容器类迭代器的用法类似。

2. 顺序容器类的迭代器的用法

下面的示例代码将 `QList<QString> list` 里的数据项逐项输出。

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QList<QString>::const_iterator i;
for (i = list.constBegin(); i != list.constEnd(); ++i)
    qDebug() << *i;
```

`constBegin()`和 `constEnd()`是用于只读迭代器的，表示起始和结束位置。

若使用反向读写迭代器，并将上面示例代码中 `list` 的数据项都改为小写，代码如下：

```
QList<QString>::reverse_iterator i;
for (i = list.rbegin(); i != list.rend(); ++i)
    *i = i->toLower();
}
```

### 3. 关联容器类的迭代器的用法

对于关联容器类 `QMap` 和 `QHash`，迭代器的“\*”操作符返回数据项的值。如果想返回键，使用 `key()`函数。对应的，用 `value()`函数返回一个项的值。

例如，下面的代码将 `QMap<int, int> map` 中所有项的键和值输出。

```
QMap<int, int> map;
...
QMap<int, int>::const_iterator i;
for (i = map.constBegin(); i != map.constEnd(); ++i)
    qDebug() << i.key() << ':' << i.value();
```

Qt API 包含很多返回值为 `QList` 或 `QStringList` 的函数，要遍历这些返回的容器，必须先复制。由于 Qt 使用了隐式共享，这样的复制并无多大开销。例如下面的代码是正确的。

```
const QList<int> sizes = splitter->sizes();
QList<int>::const_iterator i;
for (i = sizes.begin(); i != sizes.end(); ++i)
    ...
```

---

**提示** 隐式共享 (Implicit Sharing) 是对象的管理方法。一个对象被隐式共享，只是传递该对象的一个指针给使用者，而不实际复制对象数据，只有在使用者修改数据时，才实质复制共享对象给使用者。如在上面的代码中，`splitter->sizes()`返回的是一个 `QList<int>`列表对象 `sizes`，但是实际上代码并不将 `splitter->sizes()`表示的列表内容完全复制给变量 `sizes`，只是传递给它一个指针。只有当 `sizes` 发生数据修改时，才会将共享对象的数据复制给 `sizes`，这样避免了不必要的复制，减少了资源占用。

---

而下面的代码是错误的。

```
QList<int>::const_iterator i;
for (i = splitter->sizes().begin(); i != splitter->sizes().end(); ++i)
    ...
```

对于 STL 类型的迭代器，隐式共享还涉及另外一个问题，即当有一个迭代器在操作一个容器变量时，不要去复制这个容器变量。

### 3.4.3 foreach 关键字

如果只是想遍历容器中所有的项，可以使用 `foreach` 关键字。`foreach` 是 `<QtGlobal>`头文件中定

义的一个宏。使用 `foreach` 的句法是：

```
foreach (variable, container)
```

使用 `foreach` 的代码比使用迭代器更简洁。例如，使用 `foreach` 遍历一个 `QLinkedList<QString>` 的示例代码如下：

```
QLinkedList<QString> list;
...
QString str;
foreach (str, list)
    qDebug() << str;
```

用于迭代的变量也可以在 `foreach` 语句里定义，`foreach` 语句也可以使用花括号，可以使用 `break` 退出迭代，示例代码如下：

```
QLinkedList<QString> list;
...
foreach (const QString &str, list) {
    if (str.isEmpty())
        break;
    qDebug() << str;
}
```

对于 `QMap` 和 `QHash`，`foreach` 会自动访问“键——值”对里的值，所以无需调用 `values()`。如果需要访问键则可以调用 `keys()`，示例代码如下：

```
QMap<QString, int> map;
...
foreach (const QString &str, map.keys())
    qDebug() << str << ':' << map.value(str);
```

对于多值映射，可以使用两重 `foreach` 语句，示例代码如下：

```
QMultiMap<QString, int> map;
...
foreach (const QString &str, map.uniqueKeys()) {
    foreach (int i, map.values(str))
        qDebug() << str << ':' << i;
}
```

---

**注意** `foreach` 关键字遍历一个容器变量是创建了容器的一个副本，所以不能修改原来容器变量的数据项。

---

## 3.5 Qt 类库的模块

Qt 类库里大量的类根据功能分为各种模块，这些模块又分为几大类。

- Qt 基本模块（Qt Essentials）：提供了 Qt 在所有平台上的基本功能。
- Qt 附加模块（Qt Add-Ons）：实现一些特定功能的提供附加价值的模块。
- 增值模块（Value-Add Modules）：单独发布的提供额外价值的模块或工具。
- 技术预览模块（Technology Preview Modules）：一些处于开发阶段，但是可以作为技术预览使用的模块。



- Qt 工具 (Qt Tools)：帮助应用程序开发的一些工具。
- Qt 官网的 “All Modules” 页面可以查看所有这些模块的信息。

3.5.1 Qt 基本模块

Qt 基本模块是 Qt 在所有平台上的基本功能，它们在所有的开发平台和目标平台上都可用，在 Qt 5 所有版本上是源代码和二进制兼容的。这些具体的基本模块见表 3-6。

表 3-6 Qt 基本模块

模块	描述
Qt Core	其他模块都用到的核心非图形类
Qt GUI	设计 GUI 界面的基础类，包括 OpenGL
Qt Multimedia	音频、视频、摄像头和广播功能的类
Qt Multimedia Widgets	实现多媒体功能的界面组件类
Qt Network	使网络编程更简单和轻便的类
Qt QML	用于 QML 和 JavaScript 语言的类
Qt Quick	用于构建具有定制用户界面的动态应用程序的声明框架
Qt Quick Controls	创建桌面样式用户界面，基于 Qt Quick 的用户界面控件
Qt Quick Dialogs	用于 Qt Quick 的系统对话框类型
Qt Quick Layouts	用于 Qt Quick 2 界面元素的布局项
Qt SQL	使用 SQL 用于数据库操作的类
Qt Test	用于应用程序和库进行单元测试的类
Qt Widgets	用于构建 GUI 界面的 C++图形组件类

Qt Core 模块是 Qt 类库的核心，所有其他模块都依赖于此模块，如果使用 qmake 构建项目，则 Qt Core 模块是自动被加入项目的。

Qt GUI 模块提供了用于开发 GUI 应用程序的必要的类，使用 qmake 构建应用程序时，Qt GUI 模块是自动被加入项目的。如果项目中不使用 GUI 功能，则需要在项目配置文件中加入如下的一行：

```
QT -= gui
```

其他的模块一般不会被自动加入到项目，如果需要在项目中使用某个模块，则可以在项目配置中添加此模块。例如，如果需要在项目中使用 Qt Multimedia 和 Qt Multimedia Widgets 模块，需要在项目配置文件中加入如下的语句：

```
QT += multimedia multimediawidgets
```

需要在项目中使用 Qt SQL 模块，就在项目配置文件中加入如下的语句：

```
QT += sql
```

3.5.2 Qt 附加模块

Qt 附加模块可以实现一些特定目的。这些模块可能只在某些开发平台上有，或只能用于某些操作系统，或只是为了向后兼容。用户安装时可以选择性地安装这些附加模块。

表 3-7 是附加模块列表（未列出一些过时的模块，以及专门用于 QML 或 Qt Quick 的模块）。

表 3-7 Qt 附加模块

模块	描述
Active Qt	用于开发使用 ActiveX 和 COM 的 Windows 应用程序
Qt 3D	支持 2D 和 3D 渲染，提供用于开发近实时仿真系统的功能
Qt Android Extras	提供 Android 平台相关的 API
Qt Bluetooth	提供访问蓝牙硬件的功能
Qt Concurrent	提供一些类，无需使用底层的线程控制就可以编写多线程程序
Qt D-Bus	使进程间通过 D-Bus 协议通信的一些类
Qt Gamepad	使 Qt 应用程序支持游戏手柄硬件的使用
Qt Image Formats	支持附加图片格式的插件，包括 TIFF、MNG、TGA、WBMP
Qt Mac Extras	提供 macOS 平台相关的 API
Qt NFC	提供访问 NFC（近场通信）硬件的功能
Qt Positioning	提供一些类，用于通过 GPS 卫星、WiFi 等定位
Qt Print Support	提供一些用于打印控制的类
Qt Purchasing	提供一些类，在 Qt 应用程序内实现应用内购买的功能
Qt Sensors	提供访问传感器硬件的功能，以识别运动和手势
Qt Serial Bus	访问串行工业总线的功能，目前只支持 CAN 和 Modbus 协议
Qt SVG	提供显示 SVG 图片文件的类
Qt WebChannel	用于实现服务器端（QML 或 C++应用程序）与客户端（HTML/ JavaScript 或 QML 应用程序）之间的 P2P 通信
Qt WebEngine	提供类和函数，实现在应用程序中嵌入网页内容
Qt WebSockets	提供兼容于 RFC 6455 的 WebSocket 通信，WebSocket 是实现客户端程序与远端主机进行双向通信的基于 Web 的协议
Qt Windows Extras	提供 Windows 平台相关的 API
Qt XML	该模块不再维护了，应使用 Qt Core 中的 QDomStreamReader 和 QDomStreamWriter
Qt XML Patterns	提供对 XPath、XQuery、XSLT 和 XML 等的支持
Qt Charts <sup>①</sup>	用于数据显示的二维图表组件
Qt Data Visualization <sup>①</sup>	用于 3D 数据可视化显示的界面组件
Qt Virtual Keyboard <sup>①</sup>	实现不同输入法的虚拟键盘框架

① 下面的附加模块只在商业许可，或 GPLv3 许可的版本里才有。

3.5.3 增值模块

除了随 Qt5 发布的上述这些模块，还有一些模块（见表 3-8）是单独发布的，这些模块只在商业版许可的 Qt 里才有。

表 3-8 Qt 的增值模块

特性	描述
Qt for Device Creation	高效、易用、全集成的嵌入式设备应用程序开发工具，包括很多其他增值特性
Qt Quick Compiler	编译.qml 源文件生成二进制应用程序的编译器，提高载入时间和代码的安全性

3.5.4 技术预览模块

技术预览模块就是一些还处于开发和测试阶段的模块，一般技术预览模块经过几个版本的发布后会变成正式的模块。表 3-9 是 Qt 5.9 中的技术预览模块。

表 3-9 技术预览模块

模块	描述
Qt Network Authorization	基于 OAuth 协议，为应用程序提供网络账号验证的功能
Qt Speech	提供文字转语音（text-to-speech）功能支持
Qt Remote Objects	进程间或设备间通信，共享 QObject 的 API

3.5.5 Qt 工具

Qt 工具（见表 3-10）在所有支持的平台上都可以使用，用于帮助应用程序的开发和设计。

表 3-10 Qt 工具

工具	描述
Qt Designer	用于扩展 Qt Designer 的类
Qt Help	在应用程序中集成在线文档的类，实现类似于 Qt Assistant 的功能
Qt UI Tools	操作 Qt Designer 生成的窗体的类



## 第4章

# 常用界面设计组件

第2章已经介绍了设计 GUI 应用程序的基本原理和方法，在掌握了用 Qt Creator 设计应用程序的基本方法之后，要应用 Qt 编写各种应用程序，重要的就是熟练掌握 Qt 类库里各种用于界面设计或其他功能的类的使用。

Qt 类库为应用程序设计提供了大量的类，本章主要介绍设计 GUI 应用程序常用的各种界面组件的使用，这些是设计 GUI 应用程序的基础。

## 4.1 字符串与输入输出

### 4.1.1 字符串与数值之间的转换

界面设计时使用最多的组件恐怕就是 QLabel 和 QLineEdit 了，QLabel 用于显示字符串，QLineEdit 用于显示和输入字符串。这两个类都有如下的两个函数用于读取和设置显示文字。

```
QString text() const  
void setText(const QString &)
```

这两个函数都涉及到 QString 类。QString 类是 Qt 程序里经常使用的类，用于处理字符串。QString 类可以进行字符串与数字之间的转换，使用 QLineEdit 就可以实现数字量的输入与输出。

图 4-1 是实例 samp4\_1 设计时的窗体，是基于 QWidget 创建的可视化窗体。界面设计使用了布局管理，窗体上组件的布局是：上方的几个组件是一个 GridLayout，下方的 9 个组件也是一个 GridLayout，两个 GridLayout 和中间一个 VerticalSpacer 又组成一个 QVBoxLayout。

在布局设计时，要巧妙运用 VerticalSpacer 和 HorizontalSpacer，还要会设置组件的 MaximumSize 和 MinimumSize 属性，以取得期望的布局效果。例如，在图 4-1 中，两个 GridLayout 之间放了一个垂直方向的分隔，当窗体变大时，两个 GridLayout 的高度并不会发生变化；而如果不放置这个垂直分隔，两个 GridLayout 的高度都会发生变化，GridLayout 内部组件的垂直距离会发生变化。

#### 1. 普通数值与字符串之间的转换

在 UI 设计器里，选中图 4-1 中的“计算”按钮，在右键快捷菜单里单击“Go to slot...”，并在出现的对话框里选择 clicked() 信号创建槽函数，在自动生成的函数体里编写如下的代码，实现

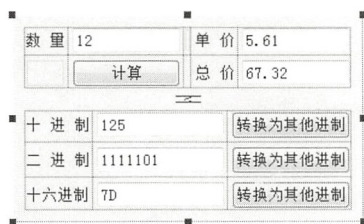


图 4-1 编辑状态的界面

从界面上分别读取数量和单价，相乘计算后将结果再显示到文本框里。

```
void Widget::on_btnCal_clicked()
{
    QString str=ui->editNum->text();//读取"数量"
    int num=str.toInt();
    str=ui->editPrice->text();//读取"单价"
    float price=str.toFloat();
    float total=num*price;
    str=str.sprintf("%.2f",total);
    ui->editTotal->setText(str);
}
```

QString 类从字符串转换为整数的函数有：

```
int      toInt(bool *ok = Q_NULLPTR, int base = 10)      const
long     toLong (bool *ok = Q_NULLPTR, int base = 10)     const
short    toShort (bool *ok = Q_NULLPTR, int base = 10)    const
uint     toUInt (bool *ok = Q_NULLPTR, int base = 10)     const
ulong    toULong (bool *ok = Q_NULLPTR, int base = 10)    const
```

这些函数如果不设置参数，缺省表示从十进制表示的字符串转换为整数；若指定整数基参数，还可以直接将二进制、十六进制字符串转换为整数。

QString 将字符串转换为浮点数的函数有：

```
double    toDouble(bool *ok = Q_NULLPTR)      const
float     toFloat (bool *ok = Q_NULLPTR)      const
```

在得到单价和数量后做相乘运算，得到计算结果变量 `total`，再将此数值显示在编辑框 `editTotal` 中。由于计算结果是浮点数，希望显示两位小数，下面 4 行语句都可以实现这个功能。

```
str=QString::number(total,'f',2);
str=QString::asprintf("%.2f",total);
str=str.setNum(total,'f',2);
str=str.sprintf("%.2f",total);
```

可以使用 QString 的静态函数 `number()` 和 `asprintf()`，也可以使用其公共函数 `setNum()` 和 `sprintf()`。QString 的 `sprintf()` 函数与 C 语言里的 `sprintf()` 函数的格式是一样的，而 `setNum()` 和 `number()` 函数使用另外一种格式定义，而且 `setNum` 和 `number` 有多个重载函数定义，可以处理各种类型的整数和浮点数，在处理整数时还可以指定进制，例如将一个整数直接转换为十六进制或二进制字符串。

## 2. 进制转换

以下是读取十进制数转换为二进制和十六进制字符串的按钮的槽函数代码：

```
void Widget::on_btnDec_clicked()
{ //读取十进制数，转换为其他进制
    QString str=ui->editDec->text();
    int val=str.toInt();//缺省为十进制
    // str=QString::number(val,16);//转换为十六进制的字符串
    str=str.setNum(val,16);//十六进制
    str=str.toUpper();
    ui->editHex->setText(str);

    str=str.setNum(val,2); //二进制
    // str=QString::number(val,2);
```

```

    ui->editBin->setText(str);
}

```

将一个整数转换为不同进制的字符串，可以使用 QString 的函数 setNum() 或静态函数 number()，它们的函数原型是：

```

QString    &setNum (int n, int base = 10)
QString    number (int n, int base = 10)

```

其中 n 是待转换的整数，base 是使用的进制，缺省为十进制，也可以指定为十六进制和二进制。下面是读取二进制字符串，然后转换为十进制和十六进制显示的按钮的槽函数代码。

```

void Widget::on_btnBin_clicked()
{ //读取二进制数，转换为其他进制的数
    QString str=ui->editBin->text(); //读取二进制字符串
    bool ok;
    int val=str.toInt(&ok,2); //以二进制数读入
    str=QString::number(val,10); //数字显示为十进制字符串
    ui->editDec->setText(str);

    str=str.setNum(val,16); //显示为十六进制
    str=str.toUpper();
    ui->editHex->setText(str);
}

```

### 4.1.2 QString 的常用功能

QString 是 Qt 编程中常用的类，除了用作数字量的输入输出之外，QString 还有很多其他功能，熟悉这些常见的功能，有助于灵活地实现字符串处理功能。

QString 存储字符串采用的是 Unicode 码，每一个字符是一个 16 位的 QChar，而不是 8 位的 char，所以 QString 处理中文字符没有问题，而且一个汉字算作是一个字符。

图 4-2 是对 QString 常用函数的测试实例 samp4\_2 的运行界面。下面在说明函数功能时，对于同名不同参数的函数，只说明某种参数下的使用实例。QString 还有很多功能函数没有在此介绍，在使用中如果遇到，可查询 Qt 的帮助文件。

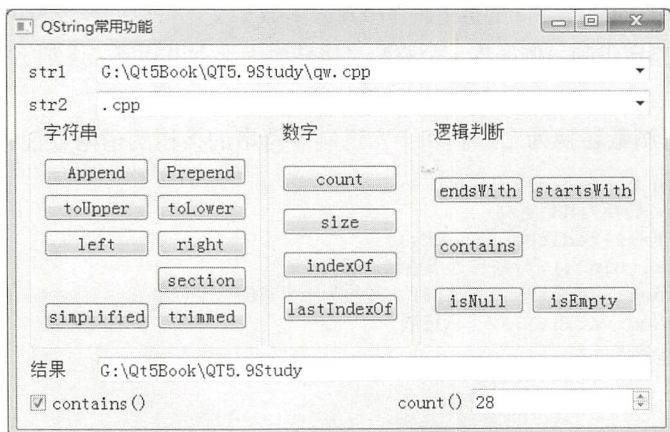


图 4-2 QString 函数功能测试实例 samp4\_2

- `append()`和 `prepend()`

`append()`在字符串的后面添加字符串, `prepend()`在字符串的前面添加字符串, 如:

```
QString str1="卖", str2="拐";
QString str3=str1;
str1.append(str2);    //str1="卖拐"
str3.prepend(str2);   //str3="拐卖"
```

- `toUpper()`和 `toLower()`

`toUpper()`将字符串内的字母全部转换为大写形式, `toLower()`将字母全部转换为小写形式, 如:

```
QString str1="Hello, World", str2;
str2=str1.toUpper();    //str1="HELLO,WORLD"
str2=str1.toLower();    //str1="hello, world"
```

- `count()`、`size()`和 `length()`

`count()`、`size()`和 `length()`都返回字符串的字符个数, 这3个函数是相同的, 但是要注意, 字符串中如果有汉字, 一个汉字算一个字符。

```
QString str1="NI 好"
N=str1.count()        //N=3
N=str1.size()         //N=3
N=str1.length()       //N=3
```

- `trimmed()`和 `simplified()`

`trimmed()`去掉字符串首尾的空格, `simplified()`不仅去掉首尾的空格, 中间连续的空格也用一个空格替换。

```
QString str1=" Are you OK? ", str2;
str2=str1.trimmed();    //str1="Are you OK? "
str2=str1.simplified(); //str1="Are you OK? "
```

- `indexOf()`和 `lastIndexOf()`

`indexOf()`函数的原型为:

```
int indexOf (const QString &str, int from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive)
const
```

其功能是在自身字符串内查找参数字符串 `str` 出现的位置, 参数 `from` 是开始查找的位置, `Qt::CaseSensitivity cs` 参数指定是否区分大小写。

`lastIndexOf()`函数则是查找某个字符串最后出现的位置。

```
QString str1="G:\\Qt5Book\\QT5.9Study\\qw.cpp";
N=str1.indexOf("5.9");    // N=13
N=str1.lastIndexOf("\\"); //N=21
```

“\\”是转义字符, 如果要查找“\\”, 需要输入“\\\\”。

- `isNull()`和 `isEmpty()`

两个函数都判断字符串是否为空, 但是稍有差别。如果一个空字符串, 只有“\0”, `isNull()`返回 `false`, 而 `isEmpty()`返回 `true`; 只有未赋值的字符串, `isNull()`才返回 `true`。

```
QString str1, str2="";
```



```

N=str1.isNull();    // N=true    未赋值字符串变量
N=str2.isNull();    // N=false   只有"\0"的字符串, 也不是 Null
N=str1.isEmpty();   // N=true
N=str2.isEmpty();   // N=true

```

QString 只要赋值, 就在字符串的末尾自动加上“\0”, 所以, 如果只是要判断字符串内容是否为空, 常用 isEmpty()。

- contains()

判断字符串内是否包含某个字符串, 可指定是否区分大小写。

```

QString str1="G:\Qt5Book\QT5.9Study\qw.cpp";
N=str1.contains(".cpp", Qt::CaseInsensitive);    // N=true, 不区分大小写
N=str1.contains(".CPP", Qt::CaseSensitive);      // N=false, 区分大小写

```

- endsWith()和 startsWith()

startsWith()判断是否以某个字符串开头, endsWith()判断是否以某个字符串结束。

```

QString str1="G:\Qt5Book\QT5.9Study\qw.cpp";
N=str1.endsWith(".cpp", Qt::CaseInsensitive);    // N=true, 不区分大小写
N=str1.endsWith(".CPP", Qt::CaseSensitive);      // N=false, 区分大小写
N=str1.startsWith("g: ");                       // N=true, 缺省为不区分大小写

```

- left()和 right()

left 表示从字符串中取左边多少个字符, right 表示从字符串中取右边多少个字符。注意, 一个汉字被当作一个字符。

```

QString str2, str1="学生姓名, 男, 1984-3-4, 汉族, 山东";
N=str1.indexOf(", ");                               // N=4, 第一个", "出现的位置
str2=str1.left(N);                                  // str2="学生姓名"
N=str1.lastIndexOf(", ");                           // N=18, 最后一个逗号的位置
str2=str1.right(str1.size()-N-1); //str2="山东", 提取最后一个逗号之后的字符串

```

- section()

section()函数的原型为:

```

QString section (const QString &sep, int start, int end = -1, SectionFlags flags =
SectionDefault) const

```

其功能是从字符串中提取以 sep 作为分隔符, 从 start 端到 end 端的字符串。

```

QString str2, str1="学生姓名, 男, 1984-3-4, 汉族, 山东";
str2=str1.section(", ", 0, 0);                       // str2="学生姓名", 第1段的编号为0
str2=str1.section(", ", 1, 1);                       // str2="男"
str2=str1.section(", ", 0, 1);                       // str2="学生姓名, 男"
str2=str1.section(", ", 4, 4);                       // str2="山东"

```

## 4.2 SpinBox 的使用

QSpinBox 用于整数的显示和输入, 一般显示十进制数, 也可以显示二进制、十六进制的数, 而且可以在显示框中增加前缀或后缀。

QDoubleSpinBox 用于浮点数的显示和输入, 可以设置显示小数位数, 也可以设置显示的前

缀和后缀。

实例 samp4\_3 演示 QSpinBox 和 QDoubleSpinBox 这两个组件的使用，图 4-3 是程序运行界面。程序功能与实例 samp4\_1 类似，但是使用 QSpinBox 和 QDoubleSpinBox 作为数字输入输出组件。

QSpinBox 和 QDoubleSpinBox 都是 QAbstractSpinBox 的子类，具有大多数相同的属性，只是参数类型不同。在 UI 设计器里进行界面设计时，就可以设置这些属性。QSpinBox 和 QDoubleSpinBox 的主要属性见表 4-1。

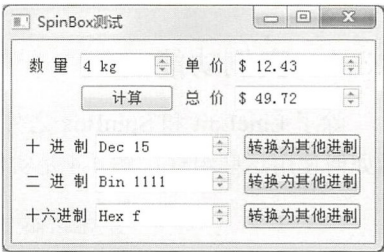


图 4-3 实例 samp4\_3 运行时界面

表 4-1 QSpinBox 和 QDoubleSpinBox 的主要属性

属性名称	描述
prefix	数字显示的前缀，例如“\$”
suffix	数字显示的后缀，例如“kg”
minimum	数值范围的最小值，如 0
maximum	数值范围的最大值，如 255
singlestep	单击右侧上下调整按钮时的单步改变值，如设置为 1，或 0.1
value	当前显示的值
displayIntegerBase	QSpinBox 特有属性，显示整数使用的进制，例如 2 就表示二进制
decimals	QDoubleSpinBox 特有属性，显示数值的小数位数，例如 2 就显示两位小数

**提示** 一个属性在类的接口中一般有一个读取函数和一个设置函数，如 QDoubleSpinBox 的 decimals 属性，读取属性值的函数为 int decimals()，设置属性值的函数为 void setDecimals(int prec)。

图 4-3 中各个 SpinBox 的类型及属性设置一目了然，不再赘述。使用 QSpinBox 和 Qdouble SpinBox 进行数值量的输入输出很方便，下面是图 4-3 中“计算”按钮和“十进制”后面的按钮的槽函数代码。

```
void Widget::on_btnCal_clicked()
{ //计算
    int num=ui->spinNum->value();
    float price=ui->spinPrice->value();
    float total=num*price;
    ui->spinTotal->setValue(total);
}
void Widget::on_btnBin_clicked()
{ //读取二进制数，以其他进制显示
    int val=ui->spinBin->value();
    ui->spinDec->setValue(val);
    ui->spinHex->setValue(val);
}
```

在使用 QSpinBox 和 QDoubleSpinBox 读取和设置数值时，无需做字符串与数值之间的转换，也无需做进制的转换，其显示效果（前缀、后缀、进制和小数位数）在设置好之后就自动按照效果进行显示，这对于数值的输入输出是非常方便的。

## 4.3 其他数值输入和显示组件

### 4.3.1 实例功能

除了 LineEdit 和 SpinBox 之外, 还有其他一些用于数值输入和显示的组件。实例 samp4\_4 演示如何使用这些组件, 图 4-4 是实例 samp4\_4 的设计界面。

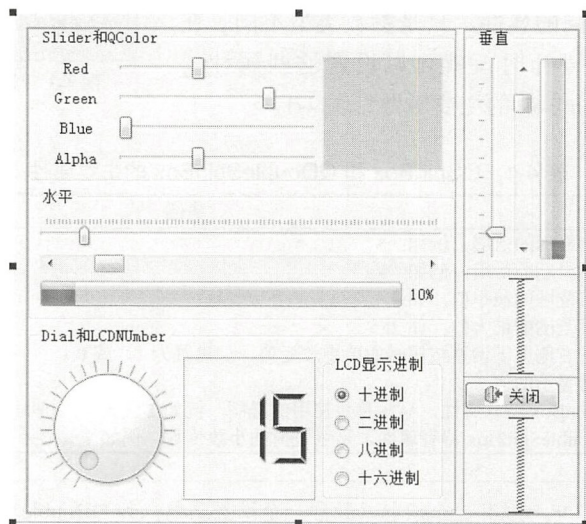


图 4-4 实例 Samp4\_4 设计界面

在这个实例中, 用到如下一些组件。

- QSlider: 滑动条, 通过滑动来设置数值, 可用于数值输入。实例中使用 4 个滑动条输入红、绿、蓝三色和 Alpha 值, 然后合成颜色, 作为一个 QTextEdit 组件的底色。
- QScrollBar: 卷滚条, 与 QSlider 功能类似, 还可以用于卷滚区域。
- QProgressBar: 进度条, 一般用于显示任务进度, 可用于数值的百分比显示。实例程序中滑动一个 Slider, 获取其值并更新 ScrollBar 和 ProgressBar。
- QDial: 表盘式数值输入组件, 通过转动表针获得输入值。
- QLCDNumber: 模仿 LCD 数字的显示组件, 可以显示整数或浮点数, 显示整数时可以不同进制显示。实例程序中转动表盘, 获得的值显示在 LCD 组件中。单击“LCD 显示进制”的 RadioButton 时, 设置 LCD 的显示进制。

### 4.3.2 各组件的主要功能和属性

#### 1. QSlider

QSlider、QScrollBar 和 Qdial 3 个组件都从 QAbstractSlider 继承而来, 有一些共有的属性。QSlider 是滑动的标尺型组件, 滑动标尺上的一个滑块可以改变值。

基类 QAbstractSlider 的主要属性包括以下几种。

- **minimum、maximum:** 设置输入范围的最小值和最大值, 例如, 用红、绿、蓝配色时, 每种基色的大小范围是 0~255, 所以设置 minimum 为 0, maximum 为 255。
- **singleStep:** 单步长, 拖动标尺上的滑块, 或按下左/右光标键时的最小变化数值。
- **pageStep:** 在 Slider 上输入焦点, 按 PgUp 或 PgDn 键时变化的数值。
- **value:** 组件的当前值, 拖动滑块时自动改变此值, 并限定在 minimum 和 maximum 定义的范围之内。
- **sliderPosition:** 滑块的位置, 若 tracking 属性设置为 true, sliderPosition 就等于 value。
- **tracking:** sliderPosition 是否等同于 value, 如果 tracking=true, 改变 value 时也同时改变 sliderPosition。
- **orientation:** Slider 的方向, 可以设置为水平或垂直。方向参数是 Qt 的枚举类型 enum Qt::Orientation, 取值包括以下两种。
  - Qt::Horizontal 水平方向
  - Qt::Vertical 垂直方向
- **invertedAppearance:** 显示方式是否反向, invertedAppearance=false 时, 水平的 Slider 由左向右数值增大, 否则反过来。
- **invertedControls:** 反向按键控制, 若 invertedControls=true, 则按下 PgUp 或 PgDn 按键时调整数值的反向相反。

属于 QSlider 的专有属性有两个, 如下。

- **tickPosition:** 标尺刻度的显示位置, 使用枚举类型 QSlider::TickPosition, 取值包括以下 6 种。
  - QSlider::NoTicks 不显示刻度
  - QSlider::TicksBothSides 标尺两侧都显示刻度
  - QSlider::TicksAbove 标尺上方显示刻度
  - QSlider::TicksBelow 标尺下方显示刻度
  - QSlider::TicksLeft 标尺左侧显示刻度
  - QSlider::TicksRight 标尺右侧显示刻度
- **tickInterval:** 标尺刻度的间隔值, 若设置为 0, 会在 singleStep 和 pageStep 之间自动选择。

## 2. QScrollBar

QScrollBar 从 QAbstractSlider 继承而来的, 具有 QAbstractSlider 的基本属性, 没有专有属性。

## 3. QDial

QDial 是仪表盘式的组件, 通过旋转表盘获得输入值。QDial 的特有的属性包括以下两种。

- **notchesVisible:** 表盘的小刻度是否可见。
- **notchTarget:** 表盘刻度间的间隔像素值。

## 4. QProgressBar

QProgressBar 的父类是 QWidget, 一般用于进度显示, 常用属性如下。

- **minimum、maximum:** 最小值和最大值。
- **value:** 当前值, 可以设定或读取当前值。



- textVisible: 是否显示文字, 文字一般是百分比表示的进度。
- orientation: 可以设置为水平或垂直方向。
- format: 显示文字的格式, “%p%” 显示百分比, “%v” 显示当前值, “%m” 显示总步数。缺省为 “%p%”。

### 5. QLCDNumber

QLCDNumber 是模拟 LCD 显示数字的组件, 可以显示整数或小数, 但就如实际的 LCD 一样, 要设定显示数字的个数。显示整数时, 还可以选择以不同进制来显示, 如十进制、二进制、十六进制。其主要属性如下。

- digitCount: 显示的数的位数, 如果是小数, 小数点也算一个数位。
- smallDecimalPoint: 是否有小数点, 如果有小数点, 就可以显示小数。
- mode: 数的显示进制, 通过调用函数 setDecMode()、setBinMode()、setOctMode()、setHexMode() 可以设置为常用的十进制、二进制、八进制、十六进制格式。
- value: 返回显示值, 浮点数。若设置为显示整数, 会自动四舍五入后得到整数, 设置为 intValue 的值。如果 smallDecimalPoint=true, 设置 value 时可以显示小数, 但是数的位数不能超过 digitCount。
- intValue: 返回显示的整数值。

例如, 若 smallDecimalPoint=true, digitCount=3, 设置 value=2.36, 则界面上 LCDNumber 组件会显示为 2.4; 若设置 value=1456.25, 则界面上 LCDNumber 组件只会显示 145。所以, 用 QLCDNumber 作为显示组件时, 应注意这些属性的配合。

## 4.3.3 实例功能的代码实现

### 1. 红绿蓝配色

使用 Red、Green、Blue 和 Alpha 4 个滑动条配色, 然后设置为旁边的 TextEdit 的底色。界面操作是在拖动滑块时就进入响应程序, 为 QSlider 的 valueChanged() 信号设计槽函数。在 UI 设计器里, 选中滑动条 SliderRed, 然后在 “Go to slot” 对话框里选择 valueChanged(int) 信号, 生成槽函数原型声明和函数体, 编写代码如下:

```
void Widget::on_SliderRed_valueChanged(int value)
{ //拖动 Red、Green、Blue 颜色滑动条时设置 textEdit 的底色
    Q_UNUSED(value);
    QColor color;
    int R=ui->SliderRed->value();
    int G=ui->SliderGreen->value();
    int B=ui->SliderBlue->value();
    int alpha=ui->SliderAlpha->value();
    color.setRgb(R,G,B,alpha); //使用 QColor 的 setRgb() 函数获得颜色
    QPalette pal=ui->textEdit->palette();
    pal.setColor(QPalette::Base,color); //设置底色
    ui->textEdit->setPalette(pal);
}
```

在这段代码里, 使用 QSlider::value() 函数获得滑动条的当前值, 然后调用 QColor::setRgb() 函

数生成颜色。QColor 的静态函数 setRgb() 定义为:

```
void QColor::setRgb(int r, int g, int b, int a = 255)
```

其中 r、g、b 是红、绿、蓝颜色值, 均在 0 在 255 之间, a 是颜色的 alpha 值, 缺省是 255, 取值范围也是 0~255。

以上代码是 SliderRed 的 valueChanged(int) 信号的槽函数代码, 其他 3 个滑动条的响应代码也与此完全相同。那么在设计程序时, 无需为其他 3 个 Slider 的 valueChanged(int) 信号再生成槽函数, 只需将它们的 valueChanged(int) 信号与槽函数 on\_SliderRed\_valueChanged() 关联即可。在窗口类 Widget 的构造函数里实现此功能, 代码如下:

```
Widget::Widget(QWidget *parent) :    QWidget(parent),    ui(new Ui::Widget)
{
    ui->setupUi(this);
    this->setLayout(ui->horizontalLayout);
    QObject::connect(ui->SliderGreen, SIGNAL(valueChanged(int)),
                     this, SLOT(on_SliderRed_valueChanged(int)));
    QObject::connect(ui->SliderBlue, SIGNAL(valueChanged(int)),
                     this, SLOT(on_SliderRed_valueChanged(int)));
    QObject::connect(ui->SliderAlpha, SIGNAL(valueChanged(int)),
                     this, SLOT(on_SliderRed_valueChanged(int)));
}
```

这样设置信号与槽的关联后, 这 4 个滑动条中任何一个的值改变时, 发射 valueChanged(int) 信号, 都调用同一个槽函数 on\_SliderRed\_valueChanged(int), 避免了重复编写代码。

## 2. LCD 值显示

滑动表盘组件的指针时, 设定 LCD 的显示值等于表盘的值。单击进制设置的 RadioButton 可以设置 LCD 的显示进制和显示数位数。为 Dial 组件的 valueChanged(int) 信号和每个 RadioButton 的 clicked() 信号生成槽函数, 编写代码实现相应的功能, 代码如下:

```
void Widget::on_dial_valueChanged(int value)
{ //设置 LCD 的显示值等于 Dial 的值
    ui->LCDDisplay->display(value);
}
void Widget::on_radioBtnDec_clicked()
{ //设置 LCD 显示十进制数
    ui->LCDDisplay->setDigitCount(3); //设置位数
    ui->LCDDisplay->setDecMode();
}
void Widget::on_radioBtnBin_clicked()
{ //设置 LCD 显示二进制数
    ui->LCDDisplay->setDigitCount(8);
    ui->LCDDisplay->setBinMode();
}
void Widget::on_radioBtnOct_clicked()
{ //设置 LCD 显示八进制数
    ui->LCDDisplay->setDigitCount(4);
    ui->LCDDisplay->setOctMode();
}
void Widget::on_radioBtnHex_clicked()
{ //设置 LCD 显示十六进制数
    ui->LCDDisplay->setDigitCount(3);
```

```
ui->LCDDisplay->setHexMode();  
}
```

## 4.4 时间日期与定时器

### 4.4.1 时间日期相关的类

时间日期是经常遇到的数据类型，Qt 中时间日期类型的类如下。

- QTime: 时间数据类型，仅表示时间，如 15:23:13。
- QDate: 日期数据类型，仅表示日期，如 2017-4-5。
- QDateTime: 日期时间数据类型，表示日期和时间，如 2017-03-23 08:12:43。

Qt 中有专门用于日期、时间编辑和显示的界面组件，介绍如下。

- QTimeEdit: 编辑和显示时间的组件。
- QDateEdit: 编辑和显示日期的组件。
- QDateTimeEdit: 编辑和显示日期时间的组件。
- QCalendarWidget: 一个用日历形式选择日期的组件。

定时器是用来处理周期性事件的一种对象，类似于硬件定时器。例如设置一个定时器的定时周期为 1000 毫秒，那么每 1000 毫秒就会发射定时器的 timeout() 信号，在信号关联的槽函数里就可以做相应的处理。Qt 中的定时器类是 QTimer，它直接从 QObject 类继承而来，不是界面组件类。

实例程序 samp4\_5 演示这些时间日期相关类的使用，其运行时界面如图 4-5 所示。



图 4-5 实例 samp4\_5 运行时界面

### 4.4.2 日期时间数据与字符串之间的转换

#### 1. 时间、日期编辑器属性设置

在图 4-5 窗体左上方的“日期时间”Groupbox 中，使用 QTimeEdit、QDateEdit、QDateTimeEdit



组件作为时间、日期、日期时间编辑器；在其右侧，各放置一个 QLineEdit 组件用于字符串显示。

QDateEdit 和 QTimeEdit 都从 QDateTimeEdit 继承而来，实现针对日期或时间的特定显示功能。实际上，QDateEdit 和 QTimeEdit 的显示功能都可以通过 QDateTimeEdit 实现，只需设置好属性即可。

QDateTimeEdit 类的主要属性的介绍如下。

- datetime: 日期时间。
- date: 日期，设置 datetime 时会自动改变 date，同样，设置 date 时，也会自动改变 datetime 里的日期。
- time: 时间，设置 datetime 时会自动改变 time，同样，设置 time 时，也会自动改变 datetime 里的时间。
- maximumDateTime、minimumDateTime: 最大、最小日期时间。
- maximumDate、minimumDate: 最大、最小日期。
- maximumTime、minimumTime: 最大、最小时间。
- currentSection: 当前输入光标所在的时间日期数据段，是枚举类型 QDateTimeEdit::Section。QDateTimeEdit 显示日期时间数据时分为多个段，单击编辑框右侧的上下按钮可修改当前段的值。如输入光标在 YearSection 段，就修改“年”的值。
- currentSectionIndex: 用序号表示的输入光标所在的段。
- calendarPopup: 是否允许弹出一个日历选择框。当取值为 true 时，右侧的输入按钮变成与 QComboBox 类似的下拉按钮，单击按钮时出现一个日历选择框，用于在日历上选择日期。对于 QTimeEdit，此属性无效。
- displayFormat: 显示格式，日期时间数据的显示格式，例如设置为“yyyy-MM-dd HH:mm:ss”，一个日期时间数据就显示为“2016-11-02 08:23:46”。

## 2. 日期时间数据的获取与转换为字符串

“读取当前日期时间”按钮的 clicked() 信号的槽函数代码如下：

```
void Dialog::on_btnGetTime_clicked()
{ // 获取当前日期时间，为三个专用编辑器设置日期时间数据，并转换为字符串
    QDateTime curDateTime=QDateTime::currentDateTime();
    ui->timeEdit->setTime(curDateTime.time());
    ui->editTime->setText(curDateTime.toString("hh:mm:ss"));
    ui->dateEdit->setDate(curDateTime.date());
    ui->editDate->setText(curDateTime.toString("yyyy-MM-dd"));
    ui->dateTimeEdit->setDateTime(curDateTime);
    ui->editDateTime->setText(curDateTime.toString("yyyy-MM-dd hh:mm:ss"));
}
```

首先用 QDateTime 类的静态函数 currentDateTime() 获取当前日期时间，并赋值给变量 curDateTime。

然后用 curDateTime 变量设置界面上 3 个日期、时间编辑器的时间或日期值，利用了 QDateTime 的 time() 和 date() 分别提取时间和日期。

将 curDateTime 表示的日期时间数据转换为字符串，然后在 QLineEdit 编辑框上显示。时间日



期转换为字符串使用了 QDateTime 的 toString()函数，分别用不同的格式显示时间、日期、日期时间。

```
ui->editTime->setText(curDateTime.toString("hh:mm:ss"));
ui->editDate->setText(curDateTime.toString("yyyy-MM-dd"));
ui->editDateTime->setText(curDateTime.toString("yyyy-MM-dd hh:mm:ss"));
```

QDateTime::toString()函数的函数原型是：

```
QString QDateTime::toString(const QString &format) const
```

它将日期时间数据按照 format 指定的格式转换为字符串。format 是一个字符串，包含一些特定的字符，表示日期或时间的各个部分，表 4-2 是用于日期时间显示的常用格式符。

表 4-2 用于日期显示的格式符及其意义

字符	意义
d	天，不补零显示，1-31
dd	天，补零显示，01-31
M	月，不补零显示，1-12
MM	月，补零显示，01-12
yy	年，两位显示，00-99
yyyy	年，4 位数字显示，如 2016
h	小时，不补零，0-23 或 1-12（如果显示 AM/PM）
hh	小时，补零 2 位显示，00-23 或 01-12（如果显示 AM/PM）
H	小时，不补零，0-23（即使显示 AM/PM）
HH	小时，补零显示，00-23（即使显示 AM/PM）
m	分钟，不补零，0-59
mm	分钟，补零显示，00-59
z	毫秒，不补零，0-999
zzz	毫秒，补零 3 位显示，000-999
AP 或 A	使用 AM/pm 显示
ap 或 a	使用 am/pm 显示

在设置日期时间显示字符串格式时，还可以使用填字符，甚至使用汉字。例如，日期显示格式可以设置为：

```
curDateTime.toString("yyyy 年 MM 月 dd 日");
```

这样得到的字符串是“2016 年 11 月 21 日”。

### 3. 字符串转换为日期时间

同样的，也可以将字符串转换为 QTime、QDate 或 QDateTime 类型，使用静态函数 QDateTime::fromString()，其函数原型为：

```
QDateTime QDateTime::fromString(const QString &string, const QString &format)
```

其中，第 1 个参数 string 是日期时间字符串形式，第 2 个参数 format 是字符串表示的格式，按照表 4-2 的格式字符定义。

在程序运行时，手工修改“日期时间”后面文本框里的日期时间字符串，单击“设置日期时间”按钮，可以将文本框里的字符串转换为 QDateTime 变量，并设置为左侧 DateTimeEdit 组件的日期时间数据，代码如下：

```
void Dialog::on_btnSetDateTime_clicked()
```

```
//字符串转换为 QDateTime
QString str=ui->editDateTime->text();
str=str.trimmed();
if (!str.isEmpty())
{
    QDateTime datetime=QDateTime::fromString(str,"yyyy-MM-dd hh:mm:ss");
    ui->dateTimeEdit->setDateTime(datetime);
}
}
```

静态函数 `QDateTime::fromString()` 将一个字符串按照格式转换为日期时间类型。程序中的代码是：

```
datetime=QDateTime::fromString(str,"yyyy-MM-dd hh:mm:ss");
```

这里将字符串 `str` 按照格式 "yyyy-MM-dd hh:mm:ss" 转换为日期时间变量，格式是指字符串 `str` 所表示的日期时间的格式。

### 4.4.3 QCalendarWidget 日历组件

图 4-5 窗体右侧是一个 `QCalendarWidget` 组件，它以日历的形式显示日期，可以用于日期选择。

`QCalendarWidget` 有一个信号 `selectionChanged()`，在日历上选择的日期变化后会发射此信号，为此信号创建槽函数，编写代码如下：

```
void Dialog::on_calendarWidget_selectionChanged()
{ //在日历上选择日期
    QDate dt=ui->calendarWidget->selectedDate();
    QString str=dt.toString("yyyy 年 M 月 d 日");
    ui->editCalendar->setText(str);
}
```

### 4.4.4 定时器的使用

Qt 中的定时器类是 `QTimer`。`QTimer` 不是一个可见的界面组件，在 UI 设计器的组件面板里找不到它。实例程序实现了一个计时器的功能，就是计算定时器开始到停止持续的时间长度，计时器是 `QTime` 类。

`QTimer` 主要的属性是 `interval`，是定时中断的周期，单位毫秒。`QTimer` 主要的信号是 `timeout()`，在定时中断时发射此信号，要想在定时中断里做出响应，这就需要编写 `timeout()` 信号的槽函数。下面是窗口类中增加的定义（省略了其他不相关的定义）：

```
class Dialog : public QDialog
{
private:
    QTimer *fTimer; //定时器
    QTime fTimeCounter;//计时器
private slots:
    void on_timer_timeout(); //定时溢出处理槽函数
};
```

这里定义了一个定时器 `fTimer`，一个计时器 `fTimeCounter`。还定义了一个槽函数 `on_timer_timeout()`，作为定时器的 `timeout()` 信号的响应槽函数。

需要在窗口类的构造函数里创建定时器，并进行信号与槽的关联。代码如下：

```
Dialog::Dialog(QWidget *parent) : QDialog(parent), ui(new Ui::Dialog)
{
    ui->setupUi(this);
    fTimer=new QTimer(this);
    fTimer->stop();
    fTimer->setInterval(1000); //设置定时周期, 单位: 毫秒
    connect(fTimer, SIGNAL(timeout()), this, SLOT(on_timer_timeout()));
}
```

槽函数 on\_timer\_timeout()的实现代码如下:

```
void Dialog::on_timer_timeout()
{ //定时器中断响应
    QTime curTime=QTime::currentTime(); //获取当前时间
    ui->LCDHour->display(curTime.hour()); //显示 小时
    ui->LCDMin->display(curTime.minute()); //显示 分钟
    ui->LCDSec->display(curTime.second()); //显示 秒
    int va=ui->progressBar->value();
    va++;
    if (va>100)
        va=0;
    ui->progressBar->setValue(va);
}
```

这段代码首先用 QTime 类的静态函数 Qtime::currentTime()获取当前时间, 然后用 QTime 的成员函数 hour()、minute()、second()分别获取小时、分钟、秒, 并在几个 LCDNumber 组件上显示。循环更新 progressBar 的值, 是为了让界面有变化, 表示定时器在运行。

设置定时器的周期, 只需调用 QTimer::setInterval()函数即可。

QTimer::start()函数用于启动定时器, 界面上的“开始”按钮代码如下:

```
void Dialog::on_btnStart_clicked()
{
    fTimer->start(); //定时器开始工作
    fTimeCounter.start(); //计时器开始工作
    ui->btnStart->setEnabled(false);
    ui->btnStop->setEnabled(true);
    ui->btnSetIntv->setEnabled(false);
}
```

计时器 fTimeCounter 执行 start()是将当前时间作为计时器的时间。

QTimer::stop()函数停止定时器, 界面上的“停止”按钮可实现这一功能, 其代码如下:

```
void Dialog::on_btnStop_clicked()
{
    fTimer->stop(); //定时器停止
    int tmMsec=fTimeCounter.elapsed(); //毫秒数
    int ms=tmMsec%1000;
    int sec=tmMsec/1000;
    QString str=QString::asprintf("流逝时间: %d 秒, %d 毫秒", sec, ms);
    ui->LabElapsTime->setText(str);
    ui->btnStart->setEnabled(true);
    ui->btnStop->setEnabled(false);
    ui->btnSetIntv->setEnabled(true);
}
```

## 4.5 QComboBox 和 QPlainTextEdit

### 4.5.1 实例功能概述

QComboBox 是下拉列表框组件类，它提供一个下拉列表供用户选择，也可以直接当作一个 QLineEdit 用作输入。QComboBox 除了显示可见下拉列表外，每个项（item，或称列表项）还可以关联一个 QVariant 类型的变量，用于存储一些不可见数据。

QPlainTextEdit 是一个多行文本编辑器，用于显示和编辑多行简单文本。实例 samp4\_6 演示 QComboBox 和 QPlainTextEdit 的使用，其运行时界面如图 4-6 所示。



图 4-6 实例 samp4\_6 运行界面

### 4.5.2 QComboBox 的使用

#### 1. 设计时属性设置

QComboBox 主要的功能是提供一个下拉列表供选择输入。在界面上放置一个 QComboBox 组件后，双击此组件，可以出现如图 4-7 所示的对话框，对 QComboBox 组件的下拉列表的项进行编辑。在图 4-7 所示的对话框中，可以进行编辑，如添加、删除、上移、下移操作，还可以设置项的图标。

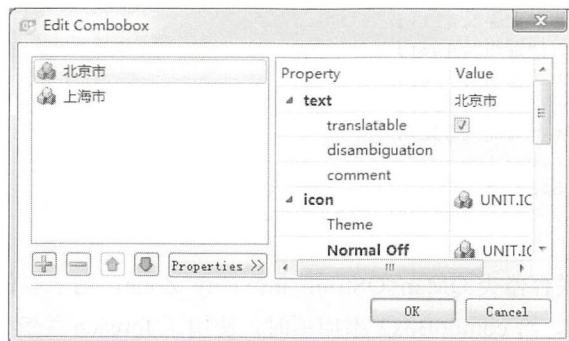


图 4-7 QComboBox 组件设计时的列表项编辑器



## 2. 用代码添加简单项

窗口上的“初始化列表”按钮初始化下拉列表框的列表内容，其代码如下：

```
void Widget::on_btnIniItems_clicked()
{ //“初始化列表”按钮
    QIcon icon;
    icon.addFile(":/images/icons/aim.ico");
    ui->comboBox->clear();
    for (int i=0;i<20;i++)
        ui->comboBox->addItem(icon,QString::asprintf("Item %d",i)); //带图标
    // ui->comboBox->addItem(QString::asprintf("Item %d",i)); //不带图标
}
```

添加一个项时可以指定一个图标，图标来源于资源文件。

addItem()用于添加一个列表项，如果只是添加字符串列表项，而且数据来源于一个 QStringList 变量，可以使用 addItems()函数，示例代码如下：

```
ui->comboBox->clear();
QStringList strList;
strList<<"北京"<<"上海"<<"天津"<<"河北省"<<"山东省"<<"山西省";
ui->comboBox->addItems(strList);
```

## 3. 添加具有用户数据的项

QComboBox::addItem()函数的两种参数的原型定义如下：

```
void addItem (const QString &text, const QVariant &userData = QVariant())
void addItem (const QIcon &icon, const QString &text, const QVariant &userData =
QVariant())
```

不管是哪一个 addItem()函数，后面都有一个可选的 QVariant 类型的参数 userData，可以利用这个变量存储用户定义数据。

界面上另一个 ComboBox 组件使用了用户数据，“初始化城市+区号”按钮的槽函数代码如下：

```
void Widget::on_btnIni2_clicked()
{//初始化具有自定义数据的 ComboBox
    QMap<QString, int> City_Zone;
    City_Zone.insert("北京",10);
    City_Zone.insert("上海",21);
    City_Zone.insert("天津",22);
    City_Zone.insert("大连",411);
    City_Zone.insert("锦州",416);
    City_Zone.insert("徐州",516);
    City_Zone.insert("福州",591);
    City_Zone.insert("青岛",532);
    ui->comboBox2->clear();
    foreach(const QString &str, City_Zone.keys())
        ui->comboBox2->addItem(str, City_Zone.value(str));
}
```

这里定义了一个关联容器类 QMap<QString, int> City\_Zone，用于存储<城市，区号>映射表。为 City\_Zone 填充数据后，给 comboBox2 添加项时，使用了 foreach 关键字遍历 City\_Zone.keys()。添加项的语句如下：

```
ui->comboBox2->addItem(str, City_Zone.value(str));
```

城市名称作为项显示的字符串，电话区号作为项关联的用户数据，但是在列表框里只能看到城市名称。

**注意** 将 City\_Zone 的内容添加到列表框之后，列表框里显示的列表项的顺序与源程序中设置 City\_Zone 的顺序不一致，因为 QMap<Key, T>容器类会自动按照 Key 排序。

#### 4. QComboBox 列表项的访问

QComboBox 存储的项是一个列表，但是 QComboBox 不提供整个列表用于访问，可以通过索引访问某个项。访问项的一些函数主要有以下几种。

- `int currentIndex()`，返回当前项的序号，第一个项的序号为 0。
- `QString currentText()`，返回当前项的文字。
- `QVariant currentData(int role = Qt::UserRole)`，返回当前项的关联数据，数据的缺省角色为 `role = Qt::UserRole`，角色的意义在 5.1 节详细介绍。
- `QString itemText(int index)`，返回指定索引号的项的文字。
- `QVariant itemData(int index, int role = Qt::UserRole)`，返回指定索引号的项的关联数据。
- `int count()`，返回项的个数。

在一个 QComboBox 组件上选择项发生变化时，会发射如下两个信号：

```
void currentIndexChanged(int index)
void currentIndexChanged(const QString &text)
```

这两个信号只是传递的参数不同，一个传递的是当前项的索引号，一个传递的当前项的文字。

为使用方便，选择为 `currentIndexChanged(const QString &text)` 信号编写槽函数。窗体上只存储字符串列表的 `comboBox` 的槽函数代码如下：

```
void Widget::on_comboBox_currentIndexChanged(const QString &arg1)
{
    ui->plainTextEdit->appendPlainText(arg1);
}
```

关联有城市区号的 `comboBox2` 的槽函数代码如下：

```
void Widget::on_comboBox2_currentIndexChanged(const QString &arg1)
{
    QString zone=ui->comboBox2->currentData().toString(); //项关联的数据
    ui->plainTextEdit->appendPlainText(arg1+":区号="+zone);
}
```

### 4.5.3 QPlainTextEdit 的使用

#### 1. QPlainTextEdit 的功能

QPlainTextEdit 是用于编辑多行文本的编辑器，可以编辑普通文本。另外，还有一个 QTextEdit 组件，是一个所见即所得的可以编辑带格式文本的组件，以 HTML 格式标记符定义文本格式。

从前面的代码已经看出，使用 `QPlainTextEdit::appendPlainText(const QString &text)` 函数就可以

向 `PlainTextEdit` 组件添加一行字符串。

`QPlainTextEdit` 提供 `cut()`、`copy()`、`paste()`、`undo()`、`redo()`、`clear()`、`selectAll()` 等标准编辑功能的槽函数, `QPlainTextEdit` 还提供一个标准的右键快捷菜单。

## 2. 逐行读取文字内容

如果要将 `QPlainTextEdit` 组件里显示的所有文字读取出来, 有一个简单的函数 `toPlainText()` 可以将全部文字内容输出为一个字符串, 其定义如下:

```
QString QPlainTextEdit::toPlainText() const
```

但是如果要逐行读取 `QPlainTextEdit` 组件里的字符串, 则稍微麻烦一点。

下面是图 4-6 窗口中“文本框内容添加到 `ComboBox`”按钮的响应代码, 它将 `plainTextEdit` 里的每一行作为一个项添加到 `comboBox` 里。

```
void Widget::on_btnToComboBox_clicked()
{ //plainTextEdit 的内容逐行添加为 comboBox 的项
    QTextDocument* doc=ui->plainTextEdit->document(); //文本对象
    int cnt=doc->blockCount(); //回车符是一个 block
    QIcon icon(":/images/icons/aim.ico");
    ui->comboBox->clear();
    for (int i=0; i<cnt;i++)
    {
        QTextBlock textLine=doc->findBlockByNumber(i); // 文本中的一段
        QString str=textLine.text();
        ui->comboBox->addItem(icon,str);
    }
}
```

`QPlainTextEdit` 的文字内容以 `QTextDocument` 类型存储, 函数 `document()` 返回这个文档对象的指针。

`QTextDocument` 是内存中的文本对象, 以文本块的方式存储, 一个文本块就是一个段落, 每个段落以回车符结束。`QTextDocument` 提供一些函数实现对文本内容的存取。

- `int blockCount()`, 获得文本块个数。
- `QTextBlock findBlockByNumber(int blockNumber)`, 读取某一个文本块, 序号从 0 开始, 至 `blockCount()-1` 结束。

一个 `document` 有多个 `TextBlock`, 从 `document` 中读取出的一个文本块类型为 `QTextBlock`, 通过 `QTextBlock::text()` 函数可以获取其纯文本文字。

## 3. 使用 `QPlainTextEdit` 自带的快捷菜单

`QPlainTextEdit` 是一个多行文字编辑框, 有自带的右键快捷菜单, 可实现常见的编辑功能。在 UI 设计器里, 选择为 `plainTextEdit` 的 `customContextMenuRequested()` 信号生成槽函数, 编写如下的代码, 就可以创建并显示 `QPlainTextEdit` 的标准快捷菜单:

```
void Widget::on_plainTextEdit_customContextMenuRequested(const QPoint &pos)
{ //创建并显示标准弹出式菜单
    QMenu* menu=ui->plainTextEdit->createStandardContextMenu();
    menu->exec(pos);
}
```

## 4.6 QListWidget 和 QToolButton

### 4.6.1 实例功能简介

Qt 中用于项 (Item) 处理的组件有两类, 一类是 Item Views, 包括 QListView、QTreeView、QTableView、QColumnView 等; 另一类是 Item Widgets, 包括 QListWidget、QTreeWidget 和 QTableWidget。

Item Views 基于模型/视图 (Model/View) 结构, 视图 (View) 与模型数据 (Model Data) 关联实现数据的显示和编辑, 模型/视图结构的使用在第 5 章详细介绍。

Item Widgets 是直接将数据存储在每一个项里, 例如, QListWidget 的每一行是一个项, QTreeWidget 的每个节点是一个项, QTableWidget 的每一个单元格是一个项。一个项存储了文字、文字的格式、自定义数据等。

Item Widgets 是 GUI 设计中常用的组件, 本节通过实例 samp4\_7 先介绍 QListWidget 以及其他一些组件的用法, 实例运行时界面如图 4-8 所示。



图 4-8 实例 Samp4\_7 运行时界面

本实例不仅介绍 QListWidget 的使用, 还包括如下一些功能的实现。

- 使用 QTabWidget 设计多页界面, 工作区右侧就是一个具有 3 个页面的 TabWidget 组件。
- 使用 QToolBox 设计多组工具箱, 工作区左侧是一个有 3 个组的 ToolBox 组件。
- 使用分隔条 (QSplitter) 设计可以左右分割的界面, 工作区的 ToolBox 和 TabWidget 之间有一个 splitter, 运行时可以分割调整两个组件的大小。
- 创建 Actions, 用 Actions 设计主工具栏, 用 Action 关联 QToolButton 按钮。
- 使用 QToolButton 按钮, 设置与 Action 关联, 设计具有下拉菜单功能的 ToolButton 按钮, 在主工具栏上添加具有下拉菜单的 ToolButton 按钮。
- 使用 QListWidget, 演示如何创建和添加项, 为项设置图标和复选框, 如何遍历列表进行选择。



- QListWidget 的主要信号 currentItemChanged()的功能，编写响应槽函数。
- 为 QListWidget 组件利用已设计的 Actions 创建自定义快捷菜单。

## 4.6.2 界面设计

### 1. 混合式界面设计

本实例的主窗口从 QMainWindow 继承而来，采用混合式界面设计。在 UI 设计器里完成的窗体界面如图 4-9 所示，与图 4-8 所示运行界面有一些区别。运行时在工具栏上增加了一个具有下拉菜单的工具栏按钮，为各个 ToolButton 按钮设置了关联的 Action。

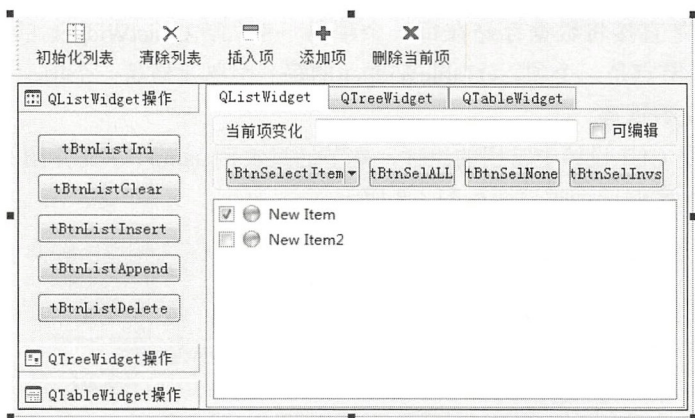


图 4-9 在 UI 设计器里完成的窗体界面

图 4-9 中界面上的按钮都使用 QToolButton 组件，在设计时只为其命名，图中按钮上显示的文字就是其 objectName。QToolButton 有一个 setDefaultAction()函数，可以使其与一个 Action 关联，按钮的文字、图标、ToolTip 都将自动设置为与关联的 Action 一致，单击一个 QToolButton 按钮就是执行 Action 的槽函数，与工具栏上的按钮一样。实际上，主工具栏上的按钮就是根据 Action 自动创建的 QToolButton 按钮。

QToolButton 还有一个 setMenu()函数，可以为其设置一个下拉式菜单，配合 QToolButton 的一些属性设置，可以有不同的下拉菜单效果。在图 4-8 中，工具栏上的“项选择”直接显示下拉菜单，而在列表框上方的“项选择”按钮，只有单击右侧的向下箭头才弹出下拉菜单，直接单击按钮会执行按钮关联的 Action 的代码。

混合式界面设计中用代码实现的部分，就是为界面上的各 ToolButton 按钮设置关联的 Action，在工具栏上动态添加一个 ToolButton，并设置其下拉菜单功能。

### 2. QToolBox 组件的设置

在 UI 设计器里设计界面时，在窗口的工作区放置一个 QToolBox 组件。

在 ToolBox 组件上调出右键快捷菜单，可以使用“Insert Page”“Delete Page”等菜单项实现分组的添加或删除。单击某个分组的标题，就可以选择为 ToolBox 组件的当前分组，在 Property Editor 中主要的属性设置如下。

- `currentIndex`, 当前分组编号, 第 1 个分组的编号是 0, 通过改变这个值, 可以选择不同的分组页面。
- `currentItemText`, 当前分组的标题。
- `currentItemName`, 当前分组的对象名称。
- `currentItemIcon`, 为当前分组设置一个图标, 显示在文字标题的左侧。

在一个 `ToolBox` 内可以放置任何界面组件, 如 `QGroupBox`、`QLineEdit`、`QPushButton` 等。在第一个分组里放置几个 `QToolButton` 按钮, 并设置为 `Grid` 布局。注意不要使用水平布局, 因为使用水平布局时, 组内的 `ToolButton` 按键都是自动向左靠齐的, 而使用 `Grid` 布局时, 自动居中。

### 3. QTabWidget 组件的设置

`QTabWidget` 是一个多页的容器类组件。在窗口上放置一个 `QTabWidget` 组件, 通过其快捷菜单的“Insert Page”“Delete Page”等菜单项实现页面的添加或删除。在 `Property Editor` 中主要的属性设置如下。

- `tabPosition`: 页标签的位置, 东、西、南、北四个方位中选择一个。
- `currentIndex`: 当前页的编号。
- `currentTabText`: 当前页的标题。
- `currentTabName`: 当前页的对象名称。
- `currentTabIcon`: 可以为当前页设置一个图标, 显示在文字标题的左侧。

### 4. 使用 QSplitter 设计分割界面

具有分割效果的典型界面是 `Windows` 的资源管理器, `QSplitter` 用于设计具有分割效果的界面, 可以左右或上下分割。

本实例主窗口两个主要的组件是 `toolBox` 和 `tabWidget`, 希望这两个组件设计为左右分割的效果。同时选择这两个组件, 单击主窗口工具栏上的“Lay Out Horizontally in Splitter”按钮, 就可以为这两个组件创建一个水平分割的布局组件 `splitter`。在主窗口构造函数里使用下面一行语句就可以使 `splitter` 填满整个工作区。

```
setCentralWidget(ui->splitter);
```

在使用分割条调整大小时, 如果不希望 `ToolBox` 的宽度变得太小而影响按钮的显示, 可以设置 `toolBox` 的 `minimumSize.Width` 属性, 设置一个最小宽度。

### 5. QListWidget 的设置

在 `TabWidget` 组件的第一个页面上放置一个 `QListWidget` 组件, 以及其他几个按钮和编辑框, 组成如图 4-9 所示的界面。`QListWidget` 是存储多个项的列表组件, 每个项是一个 `QListWidgetItem` 类型的对象。

双击 `ListWidget` 组件, 可以打开其列表项编辑器, 如图 4-10 所示。在这个编辑器里可以增加、删除、上移、下移列表项, 可以设置每个项的属性, 包括文字内容、字体、文字对齐方式、背景色、前景色等。

比较重要的是其 `flags` 属性 (如图 4-10 所示), 用于设置项的一些标记, 这些标记是枚举类型 `Qt::ItemFlag` 的具体值, 包括以下几种。

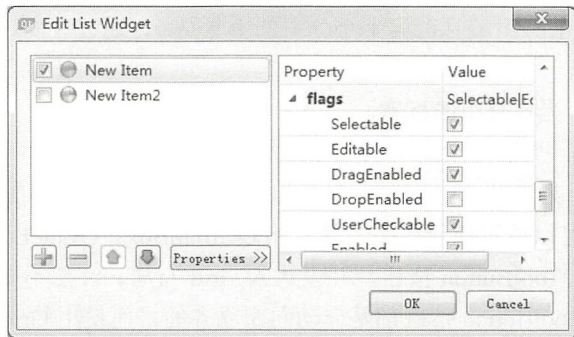


图 4-10 QListWidget 组件的列表项编辑器

- Selectable: 项是否可被选择, 对应枚举值 Qt::ItemIsSelectable。
- Editable: 项是否可被编辑, 对应枚举值 Qt::ItemIsEditable。
- DragEnabled: 项是否可以被拖动, 对应枚举值 Qt::ItemIsDragEnabled。
- DropEnabled: 项是否可以接收拖放的项, 对应枚举值 Qt::ItemIsDropEnabled。
- UserCheckable: 项是否可以被复选, 若为 true, 项前面出现一个 CheckBox, 对应枚举值 Qt::ItemIsUserCheckable。
- Enabled: 项是否被使能, 对应枚举值 Qt::ItemIsEnabled。
- Tristate: 是否允许 Check 的第三种状态, 若为 false, 则只有 checked 和 unChecked 两种状态, 对应枚举值 Qt::ItemIsAutoTristate。

在代码中设置项的 flags 属性时, 使用函数 setFlags(), 例如:

```
aItem->setFlags(Qt::ItemIsSelectable | Qt::ItemIsUserCheckable | Qt::ItemIsEnabled);
```

QListWidget 的列表项一般是在程序里动态创建, 后面会演示如何用程序完成添加、删除列表项等操作。

## 6. 创建 Action

在 2.4 节已经介绍了创建 Action, 学习用 Action 设计主菜单和主工具栏的方法。本实例也采用 Action 设计工具栏, 并且将 Action 用于 QToolButton 按钮。创建的 Action 列表如图 4-11 所示。利用这些 Action 创建主工具栏, 设计时完成的主工具栏如图 4-9 所示。

Name	Used	Text	Shortcut	Checkable	ToolTip
actListIni	<input checked="" type="checkbox"/>	初始化列表	Ctrl+I	<input type="checkbox"/>	初始化列表
actListClear	<input checked="" type="checkbox"/>	清除列表		<input type="checkbox"/>	清除列表
actListInsert	<input checked="" type="checkbox"/>	插入项	Ctrl+S	<input type="checkbox"/>	插入项
actListAppend	<input checked="" type="checkbox"/>	添加项	Ctrl+A	<input type="checkbox"/>	添加项
actListDelete	<input checked="" type="checkbox"/>	删除当前项	Del	<input type="checkbox"/>	删除当前项
<input type="checkbox"/> actSelALL	<input type="checkbox"/>	全选		<input type="checkbox"/>	全选
<input type="checkbox"/> actSelNone	<input type="checkbox"/>	全不选		<input type="checkbox"/>	全不选
<input type="checkbox"/> actSelInvs	<input type="checkbox"/>	反选		<input type="checkbox"/>	反选
actQuit	<input type="checkbox"/>	退出		<input type="checkbox"/>	退出程序
actSelPopMenu	<input type="checkbox"/>	项选择		<input type="checkbox"/>	项选择

图 4-11 本实例创建的 Action



actSelPopupMenu 用于“项选择”的 ToolButton 按钮，也就是窗口上具有下拉菜单的两个按钮。将 actSelPopupMenu 的功能设置为与 actSelInvs 完全相同，在“Signals & Slots Editor”里设置这两个 Action 关联（如图 4-12 所示），这样，执行 actSelPopupMenu 就是执行 actSelInvs。

Sender	Signal	Receiver	Slot
actSelPopupMenu	triggered()	actSelInvs	trigger()
actQuit	triggered()	MainWindow	close()

图 4-12 在 Signals 和 Slots 编辑器中设置的关联

### 4.6.3 QListWidget 的操作

#### 1. 初始化列表

actListIni 实现 listWidget 的列表项初始化，其 trigger() 信号槽函数代码如下：

```
void MainWindow::on_actListIni_triggered()
{ //初始化列表
    QListWidgetItem *aItem; //每一行是一个 QListWidgetItem
    QIcon aIcon;
    aIcon.addFile(":/images/icons/check2.ico");
    bool chk=ui->checkBoxListEditable->isChecked(); //是否可编辑

    ui->listWidget->clear();
    for (int i=0; i<10; i++)
    {
        QString str=QString::asprintf("Item %d",i);
        aItem=new QListWidgetItem();
        aItem->setText(str); //设置文字标签
        aItem->setIcon(aIcon); //设置图标
        aItem->setCheckState(Qt::Checked); //设置为选中状态
        if (chk) //可编辑，设置 flags
            aItem->setFlags(Qt::ItemIsSelectable | Qt::ItemIsEditable
|Qt::ItemIsUserCheckable |Qt::ItemIsEnabled);
        else//不可编辑，设置 flags
            aItem->setFlags(Qt::ItemIsSelectable |Qt::ItemIsUserCheckable
|Qt::ItemIsEnabled);
        ui->listWidget->addItem(aItem); //增加一个项
    }
}
```

列表框里一行是一个项，是一个 QListWidgetItem 类型的对象，向列表框添加一个项，就需要创建一个 QListWidgetItem 类型的实例 aItem，然后设置 aItem 的一些属性，再用 QListWidget::addItem() 函数将该 aItem 添加到列表框里。

QListWidgetItem 有许多函数方法，可以设置项的很多属性，如设置文字、图标、选中状态，还可以设置 flags，就是图 4-10 对话框里设置功能的代码化。

#### 2. 插入项

插入项使用 QListWidget 的 insertItem(int row, QListWidgetItem \*item) 函数，在某一行 row 的前面插入一个 QListWidgetItem 对象 item，也需要先创建这个 item，并设置好其属性。actListInsert



实现这个功能，其槽函数代码如下：

```
void MainWindow::on_actListInsert_triggered()
{ //插入一个项
    QIcon aIcon;
    aIcon.addFile(":/images/icons/check2.ico");
    bool chk=ui->chkBoxListEditable->isChecked();

    QListWidgetItem* aItem=new QListWidgetItem("New Inserted Item");
    aItem->setIcon(aIcon);
    aItem->setCheckState(Qt::Checked);
    if (chk)
        aItem->setFlags(Qt::ItemIsSelectable | Qt::ItemIsEditable
|Qt::ItemIsUserCheckable |Qt::ItemIsEnabled);
    else
        aItem->setFlags(Qt::ItemIsSelectable |Qt::ItemIsUserCheckable
|Qt::ItemIsEnabled);
    ui->listWidget->insertItem(ui->listWidget->currentRow(),aItem);
}
```

这里设置新插入的项是可选择的、能用的、可复选的。还根据界面设置，确定项是否可编辑。

### 3. 删除当前项和清空列表

删除当前项的 `actListDelete` 的槽函数代码如下：

```
void MainWindow::on_actListDelete_triggered()
{ //删除当前项
    int row=ui->listWidget->currentRow();
    QListWidgetItem* aItem=ui->listWidget->takeItem(row);
    delete aItem; //手工删除对象
}
```

`takeItem()`函数只是移除一个项，并不删除项对象，所以还需要用 `delete` 从内存中删除它。

要清空列表框的所有项，只需调用 `QListWidget::clear()`函数即可。

### 4. 遍历并选择项

界面上有“全选”“全不选”“反选”3个按钮，由3个 Action 实现，用于遍历列表框里的项，设置选择状态。用于“全选”的 `actSelALL` 的槽函数代码如下：

```
void MainWindow::on_actSelALL_triggered()
{ //全选
    int cnt=ui->listWidget->count();//个数
    for (int i=0; i<cnt; i++)
    {
        QListWidgetItem *aItem=ui->listWidget->item(i);//获取一个项
        aItem->setCheckState(Qt::Checked);//设置为选中
    }
}
```

函数 `QListWidgetItem::setCheckState(Qt::CheckState state)`设置列表项的复选状态，枚举类型 `Qt::CheckState` 有3种取值。

- `Qt::Unchecked`，不被选中。
- `Qt::PartiallyChecked`，部分选中。在目录树中的节点可能出现这种状态，比如其子节点没有全部被勾选时。

- Qt::Checked, 被选中。

### 5. QListWidget 的常用信号

QListWidget 在当前项切换时发射两个信号, 只是传递的参数不同。

- `currentRowChanged(int currentRow)`: 传递当前项的行号作为参数。
- `currentItemChanged(QListWidgetItem *current, QListWidgetItem *previous)`: 传递两个 QListWidget 对象作为参数, `current` 表示当前项, `previous` 是前一项。

当前项的内容发生变化时发射信号 `currentTextChanged(const QString &currentText)`。

为 `listWidget` 的 `currentItemChanged()` 信号编写槽函数, 代码如下:

```
void MainWindow::on_listWidget_currentItemChanged(QListWidgetItem *current, QListWidgetItem
*previous)
{ //listWidget 当前选中项发生变化
  QString str;
  if (current != NULL)
  {
    if (previous==NULL)
      str="当前项: "+current->text();
    else
      str="前一项: "+previous->text()+"; 当前项: "+current->text();
    ui->editCutItemText->setText(str);
  }
}
```

响应代码里需要判断 `current` 和 `previous` 指针是否为空, 否则使用对象时可能出现访问错误。

## 4.6.4 QToolButton 与下拉式菜单

### 1. QToolButton 关联 QAction

图 4-9 所示的界面上, 在 `ToolBox` 里放置了几个 `QToolButton` 按钮, 希望它们实现与工具栏上的按钮相同的功能; 列表框上方放置了几个 `QToolButton` 按钮, 希望它们完成列表项选择的功能。

这些功能都已经有了相应的 `Actions` 来实现, 要让 `ToolButton` 按钮实现这些功能, 无需再为其编写代码, 只需设置一个关联的 `QAction` 对象即可。

`QToolButton` 有一个函数 `setDefaultAction()`, 其函数原型为:

```
void QToolButton::setDefaultAction(QAction *action)
```

使用 `setDefaultAction()` 函数为一个 `QToolButton` 按钮设置一个 `Action` 之后, 将自动获取 `Action` 的文字、图标、`ToolTip` 等设置为按钮的相应属性。所以, 在界面设计时无需为 `QToolButton` 做过多的设置。

在主窗体类里定义一个私有函数 `setActionsForButton()`, 用来为界面上的 `QToolButton` 按钮设置关联的 `Actions`, `setActionsForButton()` 在主窗口的构造函数里调用, 其代码如下:

```
void MainWindow::setActionsForButton()
{//为 QToolButton 按钮设置 Action
  ui->tBtnListIni->setDefaultAction(ui->actListIni);
  ui->tBtnListClear->setDefaultAction(ui->actListClear);
  ui->tBtnListInsert->setDefaultAction(ui->actListInsert);
```

```

ui->tBtnListAppend->setDefaultAction(ui->actListAppend);
ui->tBtnListDelete->setDefaultAction(ui->actListDelete);

ui->tBtnSelALL->setDefaultAction(ui->actSelALL);
ui->tBtnSelNone->setDefaultAction(ui->actSelNone);
ui->tBtnSelInvs->setDefaultAction(ui->actSelInvs);
}

```

在程序启动后，界面上的 ToolButton 按钮自动根据关联的 Actions 设置其按钮文字、图标和 Tooltip。单击某个 ToolButton 按钮，就是执行了其关联的 Action 的槽函数代码。使用 Actions 集中设计功能代码，用于菜单、工具栏、ToolButton 的设计，是避免重复编写代码的一种方式。

## 2. 为 QToolButton 按钮设计下拉菜单

还可以为 QToolButton 按钮设计下拉菜单，在图 4-8 的运行窗口中，单击工具栏上的“项选择”按钮，会在按钮的下方弹出一个菜单，有 3 个菜单项用于项选择。

在主窗口类里定义一个私有函数 createSelectionPopupMenu()，并在窗口的构造函数里调用，其代码如下：

```

void MainWindow::createSelectionPopupMenu()
{ //创建下拉菜单
    QMenu* menuSelection=new QMenu(this); //创建弹出式菜单
    menuSelection->addAction(ui->actSelALL);
    menuSelection->addAction(ui->actSelNone);
    menuSelection->addAction(ui->actSelInvs);
    //listWidget 上方的 tBtnSelectItem 按钮
    ui->tBtnSelectItem->setPopupMode(QToolButton::MenuButtonPopup);
    ui->tBtnSelectItem->setToolButtonStyle(Qt::ToolButtonTextBesideIcon);
    ui->tBtnSelectItem->setDefaultAction(ui->actSelPopupMenu); //关联 Action
    ui->tBtnSelectItem->setMenu(menuSelection); //设置下拉菜单
    //工具栏上的 下拉式菜单按钮
    QToolButton *aBtn=new QToolButton(this);
    aBtn->setPopupMode(QToolButton::InstantPopup);
    aBtn->setToolButtonStyle(Qt::ToolButtonTextUnderIcon); //按钮样式
    aBtn->setDefaultAction(ui->actSelPopupMenu);
    aBtn->setMenu(menuSelection); //设置下拉菜单
    ui->mainToolBar->addWidget(aBtn); //工具栏添加按钮
    //工具栏添加分隔条，和"退出"按钮
    ui->mainToolBar->addSeparator();
    ui->mainToolBar->addAction(ui->actQuit);
}

```

这段代码首先创建一个 QMenu 对象 menuSelection，将 3 个用于选择列表项的 Action 添加作为菜单项。

tBtnSelectItem 是窗体上 ListWidget 上方具有下拉菜单的 QToolButton 按钮的名称，调用了 QToolButton 类的 4 个函数对其进行设置。

- setPopupMode(QToolButton::MenuButtonPopup)，设置其弹出菜单的模式。QToolButton::MenuButtonPopup 是一个枚举常量，这种模式下，按钮右侧有一个向下的小箭头，必须单击这个小按钮才会弹出下拉菜单，直接单击按钮会执行按钮关联的 Action，而不会弹出下拉菜单。

- `setPushButtonStyle(Qt::PushButtonTextBesideIcon)`, 设置按钮样式, 按钮标题文字在图标右侧显示。
- `setDefaultAction(ui->actSelPopupMenu)`, 设置按钮的关联 Action, `actSelPopupMenu` 与 `actSelInvs` 有信号与槽的关联, 所以, 直接单击按钮会执行“反选”的功能。
- `setMenu(menuSelection)`, 为按钮设置下拉菜单对象。

工具栏上具有下拉菜单的按钮需要动态创建。先创建 `QPushButton` 对象 `aBtn`, 同样用以上 4 个函数进行设置, 但是设置的参数稍微不同, 特别是设置弹出菜单模式为:

```
aBtn->setPopupMode(QPushButton::InstantPopup)
```

这种模式下, 工具按钮的右下角显示一个小的箭头, 单击按钮直接弹出下拉菜单, 即使为这个按钮设置了关联的 Action, 也不会执行 Action 的功能。这是这两个具有下拉菜单的 `QPushButton` 按钮的区别。

`setActionsForButton()`和 `createSelectionPopupMenu()`函数在窗口的构造函数里调用, 构造函数的完整代码如下:

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    setCentralWidget(ui->splitter);
    setActionsForButton();
    createSelectionPopupMenu();
}
```

### 4.6.5 创建右键快捷菜单

每个从 `QWidget` 继承的类都有信号 `customContextMenuRequested()`, 这个信号在鼠标右击时发射, 为此信号编写槽函数, 可以创建和运行右键快捷菜单。

本实例为 `listWidget` 组件的 `customContextMenuRequested()`信号创建槽函数, 实现快捷菜单的创建与显示, 代码如下:

```
void MainWindow::on_listWidget_customContextMenuRequested(const QPoint &pos)
{
    Q_UNUSED(pos);
    QMenu* menuList=new QMenu(this); //创建菜单
    //添加 Actions 创建菜单项
    menuList->addAction(ui->actListIni);
    menuList->addAction(ui->actListClear);
    menuList->addAction(ui->actListInsert);
    menuList->addAction(ui->actListAppend);
    menuList->addAction(ui->actListDelete);
    menuList->addSeparator();
    menuList->addAction(ui->actSelALL);
    menuList->addAction(ui->actSelNone);
    menuList->addAction(ui->actSelInvs);
    menuList->exec(QCursor::pos()); //在鼠标光标位置显示右键快捷菜单
    delete menuList;
}
```



在这段代码里，首先创建一个 QMenu 类型的对象 menuList，然后利用 QMenu 的 addAction() 方法添加已经设计的 Actions 作为菜单项。

创建完菜单后，使用 QMenu::exec() 函数显示快捷菜单。

```
menuList->exec(QCursor::pos());
```

这样会在鼠标当前位置显示弹出式菜单，静态函数 QCursor::pos() 获得鼠标光标当前位置。快捷菜单的运行效果如图 4-13 所示。

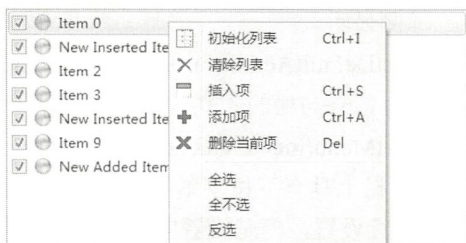


图 4-13 listWidget 组件的右键快捷菜单的运行效果

## 4.7 QTreeWidget 和 QDockWidget

### 4.7.1 实例功能概述

本节介绍 QTreeWidget、QDockWidget 的使用，以及用 QLabel 显示图片的方法。实例 samp4\_8 以 QTreeWidget 为主要组件，创建一个照片管理器，实例运行时的界面如图 4-14 所示。

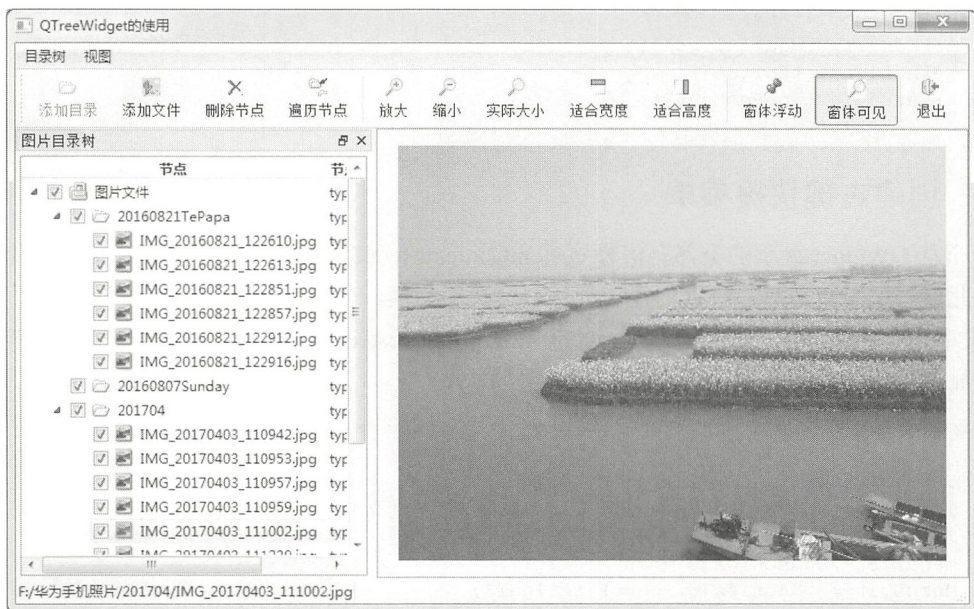


图 4-14 实例 Samp4\_8 运行时界面

这个实例主要演示如下几个组件的使用方法。

- QTreeWidget 目录树组件

QTreeWidget 类是创建和管理目录树结构的类。实例使用一个 QTreeWidget 组件管理照片目录，可以添加、删除节点，每个节点设置一个自定义类型，另外，还设置了一个自定义数据，图片节点存储完整文件名，以便单击节点时显示该图片。

- QDockWidget 停靠区域组件

QDockWidget 是可以在 QMainWindow 窗口停靠，或在桌面最上层浮动的界面组件。本实例将一个 QTreeWidget 组件放置在 QDockWidget 区域上，设置其可以在主窗口的左或右侧停靠，也可以浮动。

- QLabel 组件显示图片

右侧是一个 QScrollArea 组件，ScrollArea 上面放置一个 QLabel 组件，通过为 QLabel 设置一个 QPixmap 显示图片。通过 QPixmap 操作可进行缩放显示，包括放大、缩小、实际大小、适合宽度、适合高度等。

## 4.7.2 界面设计

### 1. 界面布局设计

实例 Samp4\_8 的主窗口从 QMainWindow 继承而来，界面采用可视化设计，程序功能主要用 Action 实现，主菜单和主工具栏也都由其实现。

工作区左侧是一个 QDockWidget 组件，在 DockWidget 上放置一个 QTreeWidget 组件，用水平布局使 treeWidget 填满停靠区。

工作区右侧是一个 QScrollArea 组件，QScrollArea 组件里放置一个 QLabel 组件，利用 QLabel 的 pixmap 属性显示图片。scrollArea 内部的组件采用水平布局，当图片较小时，Label 显示的图片可以自动居于 scrollArea 的中间；当 Label 显示的图片超过 scrollArea 可显示区域的大小后，scrollArea 会自动显示水平或垂直方向的卷滚条，用于显示更大的区域。

在主窗口构造函数里将 ScrollArea 组件设置为主窗口工作区的中心组件后，DockWidget 与 ScrollArea 之间自动出现分割条，可以分割两个组件的大小。

### 2. QDockWidget 组件属性设置

在 UI 设计器里对 DockWidget 组件的主要属性进行设置，主要属性如下。

- allowedAreas 属性，设置允许停靠区域。

由函数 setAllowedAreas(Qt::DockWidgetAreas areas)设置允许停靠区，参数 areas 是枚举类型 Qt::DockWidgetArea 的值的组合，可以设置在窗口的左、右、顶、底停靠，或所有区域都可停靠，或不允许停靠。

本实例设置为允许左侧和右侧停靠。

- features 属性，设置停靠区组件的特性。

由 setFeatures(DockWidgetFeatures features)函数设置停靠区组件的特性，参数 features 是枚举类型 QDockWidget::DockWidgetFeature 的值的组合，枚举值如下。

- QDockWidget::DockWidgetClosable: 停靠区可关闭。
- QDockWidget::DockWidgetMovable: 停靠区可移动。
- QDockWidget::DockWidgetFloatable: 停靠区可浮动。
- QDockWidget::DockWidgetVerticalTitleBar: 在停靠区左侧显示垂直标题栏。
- QDockWidget::AllDockWidgetFeatures: 使用以上所有特征。
- QDockWidget::NoDockWidgetFeatures: 不能停靠、移动和关闭。

本实例设置为可关闭、可停靠、可浮动。

### 3. QTreeWidgetItem 组件的设置

在 UI 设计器里，双击界面上的 QTreeWidgetItem 组件，可以打开图 4-15 所示的设计器，设计器有两页，可分别对 Columns 和 Items 进行设计。



图 4-15 QTreeWidgetItem 组件的设计器 (Items 页面)

Columns 页用于设计目录树的列，目录树可以有多个列。在设计器里可以添加、删除、移动列，设置列的文字、字体、前景色、背景色、文字对齐方式、图标等。本实例设置了两个列，标题分别为“节点”和“节点类型”。

Items 页面用于设计目录树的节点，可对每个节点设置属性，如文字、字体、图标等，特别是 flags 属性，可以设置节点是否可选、是否可编辑、是否有 CheckBox 等，还可以设置节点的 CheckState。在图 4-15 下方有一组按钮可以新增节点、新增下级节点、删除节点、改变节点级别、平级移动节点等。

使用设计器设计目录树的列和节点，适用于创建固定结构的目录树，但是目录树一般是根据内容动态创建的，需要运用代码实现节点的创建。

### 4. Action 设计

本例的功能代码大多采用 Action 实现，在 Action Editor 里设计 Action，然后利用 Action 设计主菜单和主工具栏。设计完成的 Action 如图 4-16 所示。

Name	Used	Text	Shortcut	Checkable	ToolTip
actAddFolder	<input checked="" type="checkbox"/>	添加目录...	Ctrl+F	<input type="checkbox"/>	添加目录
actAddFiles	<input checked="" type="checkbox"/>	添加文件...	Ctrl+N	<input type="checkbox"/>	添加文件
actZoomIn	<input checked="" type="checkbox"/>	放大	Ctrl+I	<input type="checkbox"/>	放大图片
actZoomOut	<input checked="" type="checkbox"/>	缩小	Ctrl+O	<input type="checkbox"/>	缩小图片
actZoomRealSize	<input checked="" type="checkbox"/>	实际大小		<input type="checkbox"/>	图片实际大小显示
actZoomFitW	<input checked="" type="checkbox"/>	适合宽度	Ctrl+W	<input type="checkbox"/>	适合宽度显示
actDeleteItem	<input checked="" type="checkbox"/>	删除节点		<input type="checkbox"/>	删除节点
actQuit	<input checked="" type="checkbox"/>	退出		<input type="checkbox"/>	退出本系统
actZoomFitH	<input checked="" type="checkbox"/>	适合高度	Ctrl+H	<input type="checkbox"/>	适合高度
actScanItems	<input checked="" type="checkbox"/>	遍历节点		<input type="checkbox"/>	遍历节点
actDockVisible	<input checked="" type="checkbox"/>	窗体可见		<input checked="" type="checkbox"/>	窗体可见
actDockFloat	<input checked="" type="checkbox"/>	窗体浮动		<input checked="" type="checkbox"/>	窗体浮动

图 4-16 设计的 Action



### 4.7.3 QTreeWidget 操作

#### 1. 本实例的目录树节点操作规则

本实例的目录树节点操作定义如下一些规则。

- 将目录树的节点分为 3 种类型，顶层节点、分组节点和图片节点。
- 窗口创建时初始化目录树，它只有一个顶层节点，这个顶层节点不能被删除，而且不允许再新建顶层节点。
- 顶层节点下允许添加分组节点和图片节点。
- 分组节点下可以添加分组节点和图片节点，分组节点的级数无限制。
- 图片节点是终端节点，可以在图片节点同级再添加图片节点。
- 每个节点创建时设置其类型信息，图片节点存储其完整文件名作为自定义数据。
- 单击一个图片文件节点时，显示其关联文件的图片。

为便于后面说明代码的实现，将主窗口类 `MainWindow` 中增加的自定义内容先列出来，代码如下。这些枚举类型、变量和函数的功能在后面再具体介绍。

```
class MainWindow : public QMainWindow
{
private:
//枚举类型 treeItemType, 创建节点时用作 type 参数, 自定义类型必须大于 1000
enum treeItemType{itTopItem=1001, itGroupItem, itImageItem};
enum treeColNum{colItem=0, colItemType=1}; //目录树列的编号

QLabel *LabFileName; //用于状态栏文件名显示
QPixmap curPixmap; //当前的图片
float pixRatio; //当前图片缩放比例

void iniTree();//目录树初始化
void addFolderItem(QTreeWidgetItem *parItem, QString dirName);//添加目录
QString getFinalFolderName(const QString &fullPathName); //提取目录名称
void addImageItem(QTreeWidgetItem *parItem,QString aFilename);//添加图片
void displayImage(QTreeWidgetItem *item); //显示一个图片节点的图片
void changeItemCaption(QTreeWidgetItem *item); //遍历改变节点标题
};
```

#### 2. 目录树初始化添加顶层节点

主窗口 `MainWindow` 的构造函数会调用自定义函数 `iniTree()`，对目录树进行初始化，窗口构造函数和 `iniTree()`代码如下：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    LabFileName=new QLabel("");
    ui->statusBar->addWidget(LabFileName);
    this->setCentralWidget(ui->scrollArea);
    iniTree();//初始化目录树
}

void MainWindow::iniTree()
```



```

{ //初始化目录树
    QString dataStr=""; // Item 的 Data 存储的 string
    ui->treeFiles->clear();
    QIcon icon;
    icon.addFile(":/images/icons/15.ico");

    QTreeWidgetItem* item=new QTreeWidgetItem(MainWindow::itTopItem);
    item->setIcon(MainWindow::colItem,icon); //第1列的图标
    item->setText(MainWindow::colItem,"图片文件"); //第1列的文字
    item->setText(MainWindow::colItemType,"type=itTopItem"); //第2列
    item->setFlags(Qt::ItemIsSelectable | Qt::ItemIsUserCheckable | Qt::ItemIsEnabled
| Qt::ItemIsAutoTristate);
    item->setCheckState(colItem,Qt::Checked);

    item->setData(MainWindow::colItem,Qt::UserRole,QVariant(dataStr));
    ui->treeFiles->addTopLevelItem(item); //添加顶层节点
}

```

QTreeWidgetItem 的每个节点都是一个 QTreeWidgetItem 对象，添加一个节点前需先创建它，并做好相关设置。

创建节点的语句是：

```
item=new QTreeWidgetItem(MainWindow::itTopItem);
```

传递了一个枚举常量 MainWindow::itTopItem 作为构造函数的参数，表示节点的类型。在构造函数里传递一个类型值之后，就可以用 QTreeWidgetItem::type() 返回这个节点的类型值。

itTopItem 是在 MainWindow 里定义的枚举类型 treeItemType 的一个常量值。枚举类型 treeItemType 定义了节点的类型，自定义的节点类型值必须大于 1000。

QTreeWidgetItem 的 setIcon() 和 setText() 都需要传递一个列号作为参数，指定对哪个列进行设置。列号可以直接用数字，但是为了便于理解代码和统一修改，在 MainWindow 里定义了枚举类型 treeColNum，colItem 表示第 1 列，colItemType 表示第 2 列。

setFlags() 函数设置节点的一些属性标记，是 Qt::ItemFlag 枚举类型常量的组合。

setData() 函数为节点的某一列设置一个角色数据，setData() 函数原型为：

```
void QTreeWidgetItem::setData(int column, int role, const QVariant &value)
```

其中，column 是列号，role 是角色的值，value 是一个 QVariant 类型的数。

代码中设置节点数据的语句是：

```
item->setData(MainWindow::colItem,Qt::UserRole,QVariant(dataStr));
```

它为节点的第 1 列，角色 Qt::UserRole，设置了一个字符串数据 dataStr。Qt::UserRole 是枚举类型 Qt::ItemDataRole 中一个预定义的值，关于节点的角色和 Qt::ItemDataRole 在 5.1 节详细介绍。

创建并设置好节点后，用 QTreeWidgetItem::addTopLevelItem() 函数将节点作为顶层节点添加到目录树。

### 3. 添加目录节点

actAddFolder 是用于添加组节点的 Action，当目录树上的当前节点类型是 itTopItem 或 itGroupItem 类型时，才可以添加组节点。actAddFolder 的 triggered() 信号的槽函数，以及相关自定

义函数的代码如下：

```
void MainWindow::on_actAddFolder_triggered()
{
    // 添加组节点
    QString dir=QFileDialog::getExistingDirectory();//选择目录
    if (!dir.isEmpty())
    {
        QTreeWidgetItem *parItem=ui->treeFiles->currentItem(); //当前节点
        addFolderItem(parItem,dir); //在父节点下面添加一个组节点
    }
}

void MainWindow::addFolderItem(QTreeWidgetItem *parItem, QString dirName)
{
    //添加一个组节点
    QIcon icon(":/images/icons/open3.bmp");
    QString NodeText=getFinalFolderName(dirName); //获得最后的文件夹名称

    QTreeWidgetItem *item;
    item=new QTreeWidgetItem(MainWindow::itGroupItem); //节点类型 itGroupItem
    item->setIcon(colItem,icon);
    item->setText(colItem,NodeText);
    item->setText(colItemType,"type=itGroupItem");
    item->setFlags(Qt::ItemIsSelectable | Qt::ItemIsUserCheckable | Qt::ItemIsEnabled |
Qt::ItemIsAutoTristate);
    item->setCheckState(colItem,Qt::Checked);
    item->setData(colItem,Qt::UserRole,QVariant(dirName));
    parItem->addChild(item); //在父节点下面添加子节点
}

QString MainWindow::getFinalFolderName(const QString &fullPathName)
{
    //从一个完整目录名称里，获得最后的文件夹名称
    int cnt=fullPathName.length();
    int i=fullPathName.lastIndexOf("/");
    QString str=fullPathName.right(cnt-i-1);
    return str;
}
```

actAddFolder 的槽函数首先用文件对话框获取一个目录名称，再获取目录树的当前节点，然后调用自定义函数 addFolderItem()添加一个组节点，新添加的节点将会作为当前节点的子节点。

addFolderItem()函数根据传递来的父节点 parItem 和目录全称 dirName，创建并添加节点。首先用自定义函数 getFinalFolderName()获取目录全称的最后一级的文件夹名称，这个文件夹名称将作为新建节点的标题；然后创建一个节点，创建时设置其节点类型为 itGroupItem，表示分组节点，再设置属性和关联数据，关联数据就是目录的全路径字符串；最后调用 QTreeWidgetItem:: addChild()函数，将创建的节点作为父节点的一个子节点添加到目录树。

#### 4. 添加图片文件节点

actAddFiles 是添加图片文件节点的 Action，目录树的当前节点为任何类型时这个 Action 都可用。actAddFiles 的槽函数，以及相关自定义函数的代码如下：

```
void MainWindow::on_actAddFiles_triggered()
{
    //添加图片文件节点
    QStringList files=QFileDialog::getOpenFileNames(this,
```

```

        "选择一个或多个文件","", "Images (*.jpg)");
    if (files.isEmpty())
        return;
    QTreeWidgetItem *parItem, *item;
    item=ui->treeFiles->currentItem();
    if (item->type()==itImageItem) //当前节点是图片节点
        parItem=item->parent();
    else
        parItem=item;

    for (int i = 0; i < files.size(); ++i)
    {
        QString aFilename=files.at(i); //得到一个文件名
        addImageItem(parItem,aFilename); //添加一个图片节点
    }
}

void MainWindow::addImageItem(QTreeWidgetItem *parItem, QString aFilename)
{ //添加一个图片文件节点
    QIcon icon(":/images/icons/3l.ico");
    QString NodeText=getFinalFolderName(aFilename); //获得最后的文件名称
    QTreeWidgetItem *item;
    item=new QTreeWidgetItem(MainWindow::itImageItem); //节点类型 itImageItem
    item->setIcon(colItem,icon);
    item->setText(colItem,NodeText);
    item->setText(colItemType, "type=itImageItem");
    item->setFlags(Qt::ItemIsSelectable | Qt::ItemIsUserCheckable | Qt::ItemIsEnabled |
Qt::ItemIsAutoTristate);
    item->setCheckState(colItem,Qt::Checked);
    item->setData(colItem,Qt::UserRole,QVariant(aFilename)); //完整文件名称
    parItem->addChild(item); //在父节点下面添加子节点
}

```

actAddFiles 的槽函数首先用 QFileDialog::getOpenFileNames(), 获取图片文件列表, 通过 QTreeWidgetItem::currentItem()函数获得目录树的当前节点 item。

item->type()将返回节点的类型, 也就是创建节点时传递给构造函数的那个参数。如果当前节点类型是图片节点(itImageItem), 就使用当前节点的父节点, 作为将要添加的图片节点的父节点, 否则就用当前节点作为父节点。

然后遍历所选图片文件列表, 调用自定义函数 addImageItem()逐一添加图片节点到父节点下。

addImageItem()根据图片文件名称, 创建一个节点并添加到父节点下面, 在使用 setData()设置节点数据时, 将图片带路径的文件名 aFilename 作为节点的数据, 这个数据在单击节点打开图片时会用到。

### 5. 当前节点变化后的响应

目录树上当前节点变化时, 会发射 currentItemChanged()信号, 为此信号创建槽函数, 实现当前节点类型判断、几个 Action 的使能控制、显示图片等功能, 代码如下:

```

void MainWindow::on_treeFiles_currentItemChanged(QTreeWidgetItem *current,
QTreeWidgetItem *previous)
{ //当前节点变化时的处理
    Q_UNUSED(previous);
    if (current==NULL)

```

```

    return;
int var=current->type();//节点的类型
switch(var)
{
    case itTopItem: //顶层节点
        ui->actAddFolder->setEnabled(true);
        ui->actAddFiles->setEnabled(true);
        ui->actDeleteItem->setEnabled(false); //顶层节点不能删除
        break;

    case itGroupItem: //组节点
        ui->actAddFolder->setEnabled(true);
        ui->actAddFiles->setEnabled(true);
        ui->actDeleteItem->setEnabled(true);
        break;

    case itImageItem: //图片文件节点
        ui->actAddFolder->setEnabled(false); //图片节点下不能添加目录节点
        ui->actAddFiles->setEnabled(true);
        ui->actDeleteItem->setEnabled(true);
        displayImage(current); //显示图片
        break;
}
}
}

```

`current` 是变化后的当前节点，通过 `current->type()` 获得当前节点的类型，根据节点类型控制界面上 3 个 Action 的使能状态。如果是图片文件节点，还调用 `displayImage()` 函数显示节点关联的图片。

`displayImage()` 函数的功能实现在后面介绍 `QLabel` 图片显示的部分会详细说明。

## 6. 删除节点

除了顶层节点之外，选中一个节点后也可以删除它。`actDeleteItem` 实现节点删除，其代码如下：

```

void MainWindow::on_actDeleteItem_triggered()
{//删除节点
    QTreeWidgetItem* item =ui->treeFiles->currentItem(); //当前节点
    QTreeWidgetItem* parItem=item->parent(); //父节点
    parItem->removeChild(item); //移除一个子节点，并不会删除
    delete item;
}

```

一个节点不能移除自己，所以需要获取其父节点，使用父节点的 `removeChild()` 函数来移除自己。`removeChild()` 移除一个节点，但是不从内存中删除它，所以还需调用 `delete`。

若要删除顶层节点，则使用 `QTreeWidget::takeTopLevelItem(int index)` 函数。

## 7. 节点的遍历

目录树的节点都是 `QTreeWidgetItem` 类，可以嵌套多层。有时需要在目录树中遍历所有节点，比如按条件查找某些节点、统一修改节点的标题等。遍历节点需要用到 `QTreeWidgetItem` 类的一些关键函数，还需要设计嵌套函数。

`actScanItems` 实现工具栏上“遍历节点”的功能，其槽函数及相关自定义函数代码如下：

```

void MainWindow::on_actScanItems_triggered()

```



```

{ //遍历节点
    for (int i=0;i<ui->treeFiles->topLevelItemCount();i++)
    {
        QTreeWidgetItem *item=ui->treeFiles->topLevelItem(i); //顶层节点
        changeItemCaption(item); //更改节点标题
    }
}

void MainWindow::changeItemCaption(QTreeWidgetItem *item)
{ //改变节点的标题文字
    QString str="*"+ item->text(colItem); //节点标题前加"*"
    item->setText(colItem,str);
    if (item->childCount()>0)
        for (int i=0;i< item->childCount();i++) //遍历子节点
            changeItemCaption(item->child(i)); //调用自己,可重入的函数
}

```

QTreeWidgetItem 组件的顶层节点没有父节点,要访问所有顶层节点,用到两个函数。

- int topLevelItemCount(): 返回顶层节点个数。
- QTreeWidgetItem\* topLevelItem(int index): 返回序号为 index 的顶层节点。

on\_actScanItems\_triggered()函数的 for 循环访问所有顶层节点,获取一个顶层节点 item 之后,调用 changeItemCaption(item)改变这个节点及其所有子节点的标题。

changeItemCaption(QTreeWidgetItem \*item)是一个嵌套调用函数,即在这个函数里还会调用它自己。它的前两行更改传递来的节点 item 的标题,即在标题前加星号。后面的代码根据 item->childCount()是否大于 0,判断这个节点是否有子节点,如果有子节点,则在后面的 for 循环里,逐一获取,并作为参数调用 changeItemCaption()函数。

#### 4.7.4 QLabel 和 QPixmap 显示图片

##### 1. 显示节点关联的图片

在目录树上单击一个节点后,如果其类型为图片节点 (itImageItem),就会调用 displayImage(QTreeWidgetItem \*item)函数显示节点的图片,当前节点作为函数的传递参数。displayImage()函数的代码如下:

```

void MainWindow::displayImage(QTreeWidgetItem *item)
{ //显示图片,节点 item 存储了图片文件名
    QString filename=item->data(colItem,Qt::UserRole).toString();//文件名
    LabFileName->setText(filename);
    curPixmap.load(filename);
    on_actZoomFitH_triggered(); //自动适应高度显示
}

```

QTreeWidgetItem::data()返回节点存储的数据,也就是用 setData()设置的数据。前面在添加图片节点时,将文件名的带路径全名存储为节点的数据,这里的第一行语句就可以获得节点存储的图片文件全名。

curPixmap 是在 MainWindow 中定义的一个 QPixmap 类型的变量,用于操作图片。QPixmap::load(QString &fileName)直接将一个图片文件载入。

最后调用函数 `on_actZoomFitH_triggered()` 显示图片，这是 `actZoomFitH` 的槽函数，以适应高度的形式显示图片。

## 2. 图片缩放与显示

有几个 Action 实现图片的缩放显示，包括适合宽度、适合高度、放大、缩小、实际大小，部分槽函数代码如下：

```
void MainWindow::on_actZoomFitH_triggered()
{ //适应高度显示图片
    int H=ui->scrollArea->height();
    int realH=curPixmap.height();
    pixRatio=float(H)/realH; //当前显示比例，必须转换为浮点数
    QPixmap pix=curPixmap.scaledToHeight(H-30); //图片缩放到指定高度
    ui->LabPicture->setPixmap(pix);
}
void MainWindow::on_actZoomIn_triggered()
{ //放大显示
    pixRatio=pixRatio*1.2;
    int w=pixRatio*curPixmap.width();
    int h=pixRatio*curPixmap.height();
    QPixmap pix=curPixmap.scaled(w,h);
    ui->LabPicture->setPixmap(pix);
}
void MainWindow::on_actZoomRealSize_triggered()
{ //实际大小显示
    pixRatio=1;
    ui->LabPicture->setPixmap(curPixmap);
}
```

QPixmap 存储图片数据，可以缩放图片，有以下几个函数。

- QPixmap `scaledToHeight(int height)`: 返回一个缩放后的图片的副本，图片缩放到一个高度 `height`。
- QPixmap `scaledToWidth(int width)`: 返回一个缩放后的图片的副本，图片缩放到一个宽度 `width`。
- QPixmap `scaled(int width, int height)`: 返回一个缩放后的图片的副本，图片缩放到宽度 `width` 和高度 `height`，缺省为不保持比例。

变量 `curPixmap` 保存了图片的原始副本，要缩放只需调用 `curPixmap` 的相应函数，返回缩放后的图片副本。

在界面上的一个标签 `LabPicture` 上显示图片，使用了 `QLabel` 的 `setPixmap(const QPixmap &)` 函数。

### 4.7.5 QDockWidget 的操作

程序运行时，主窗口上的 `DockWidget` 组件可以被拖动，在主窗口的左、右两侧停靠，或在桌面上浮动。工具栏上“窗体浮动”和“窗口可见”两个按钮可以用代码控制停靠区是否浮动、是否可见，其代码如下：

```
void MainWindow::on_actDockVisible_toggled(bool arg1)
```

```

{ // 停靠区的可见性
    ui->dockWidget->setVisible(arg1);
}
void MainWindow::on_actDockFloat_triggered(bool checked)
{ // 停靠区浮动性
    ui->dockWidget->setFloating(checked);
}

```

单击 DockWidget 组件标题栏的关闭按钮时，会隐藏停靠区并发射信号 visibilityChanged(bool); 当拖动 DockWidget 组件，使其浮动或停靠时，会发射信号 topLevelChanged(bool)。为这两个信号编写槽函数，可更新两个 Actions 的状态。

```

void MainWindow::on_dockWidget_visibilityChanged(bool visible)
{ // 停靠区可见性变化
    ui->actDockVisible->setChecked(visible);
}
void MainWindow::on_dockWidget_topLevelChanged(bool topLevel)
{ // 停靠区浮动性变化
    ui->actDockFloat->setChecked(topLevel);
}

```

## 4.8 QTableWidget 的使用

### 4.8.1 QTableWidget 概述

QTableWidget 是 Qt 中的表格组件类。在窗体上放置一个 QTableWidget 组件后，可以在 Property Editor 里对其进行属性设置，双击这个组件，可以打开一个编辑器，对其 Colum、Row 和 Item 进行编辑。一个 QTableWidget 组件的界面基本结构如图 4-17 所示，这个表格设置为 6 行 5 列。

表格的第 1 行称为行表头，用于设置每一列的标题，第 1 列称为列表头，可以设置其标题，但一般使用缺省的标题，即为行号。行表头和列表头一般是不可编辑的。

除了行表头和列表头之外的表格区域是内容区，内容区是规则的网格状，如同一个二维数组，每个网格单元称为一个单元格。每个单元格有一个行号、列号，图 4-17 表示了行号、列号的变化规律。

在 QTableWidget 表格中，每一个单元格是一个 QTableWidgetItem 对象，可以设置文字内容、字体、前景色、背景色、图标，也可以设置编辑和显示标记。每个单元格还可以存储一个 QVariant 数据，用于设置用户自定义数据。

列1	列2	列3	列4	列5
1 0行, 0列	0行, 1列			0行, 4列
2 1行, 0列				
3 2行, 0列				
4				
5				
6 5行, 0列				5行, 4列

图 4-17 一个 QTableWidget 表格的基本结构和工作区的行、列索引号

实例 samp4\_9 以 QTableWidget 为主要组件，演示 QTableWidget 一些主要操作的实现。实例运行时的界面如图 4-18 所示，该实例将演示以下功能的实现方法。

- 设置表格的列数和行数，设置表头的文字、格式等。
- 初始化表格数据，设置一批实例数据填充到表格里。



图 4-18 实例 Samp4\_9 的运行时界面

- 插入行、添加行、删除当前行的操作。
- 遍历表格所有单元格，读取表格内容到一个 QPlainTextEdit 里，表格的一行数据作为一行文本。
- 表格上选择的当前单元格变化时，在状态栏显示单元格存储的信息。

## 4.8.2 界面设计与初始化

Samp4\_9 的主窗体从 QMainWindow 继承而来。在图 4-18 所示的窗口上，一个 QTableWidget 组件和一个 QPlainTextEdit 组件组成上下分割布局 splitter。左侧的按钮都放在一个 QGroupBox 组件里，采用 Grid 布局，然后将 groupBox 与 splitter 采用左右分割布局。这是一个典型的三区分割的布局。

在主窗口类 MainWindow 里自定义了一些变量和函数，用于后面的代码实现，下面是在 MainWindow 的 private 部分自定义的变量和函数：

```
private:
// 自定义单元格 Type 的类型，在创建单元格的 item 时使用
enum CellType{ctName=1000,ctSex,ctBirth,ctNation,ctPartyM,ctScore};
// 各字段在表格中的列号
enum FieldColNum{colName=0,colSex,colBirth,
                  colNation,colScore,colPartyM};
QLabel *labCellIndex; //状态栏上用于显示单元格的行号、列号
QLabel *labCellType; //状态栏上用于显示单元格的 type
QLabel *labStudID; //状态栏上用于显示学号
void createItemsARow(int rowNo,QString Name,QString Sex,QDate birth,
                     QString Nation,bool isPM,int score); //为某一行创建 items
```

枚举类型 CellType 是用来表示单元格类型的，在创建单元格时使用。

枚举类型 FieldColNum 用枚举常量表示各字段在表格中的列号。



在 MainWindow 的构造函数里对界面进行初始化，代码如下：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    setCentralWidget(ui->splitterMain);
    //状态栏初始化创建
    labCellIndex = new QLabel("当前单元格坐标: ",this);
    labCellIndex->setMinimumWidth(250);
    labCellType=new QLabel("当前单元格类型: ",this);
    labCellType->setMinimumWidth(200);
    labStudID=new QLabel("学生 ID: ",this);
    labStudID->setMinimumWidth(200);
    ui->statusBar->addWidget(labCellIndex); //加到状态栏
    ui->statusBar->addWidget(labCellType);
    ui->statusBar->addWidget(labStudID);
}
```

### 4.8.3 QTableWidgetItem 操作

#### 1. 设置表头

界面上的“设置表头”按钮实现对表头的设置，其 clicked()信号的槽函数代码如下：

```
void MainWindow::on_btnSetHeader_clicked()
{ //设置表头
    QTableWidgetItem *headerItem;
    QStringList headerText;
    headerText<<"姓 名"<<"性 别"<<"出生日期"<<"民 族"<<"分数"<<"是否党员";
    // ui->tableInfo->setHorizontalHeaderLabels(headerText);
    ui->tableInfo->setColumnCount(headerText.count());
    for (int i=0;i<ui->tableInfo->columnCount();i++)
    {
        headerItem=new QTableWidgetItem(headerText.at(i));
        QFont font=headerItem->font();
        font.setBold(true); //设置为粗体
        font.setPointSize(12); //字体大小
        headerItem->setTextColor(Qt::red); //字体颜色
        headerItem->setFont(font); //设置字体
        ui->tableInfo->setHorizontalHeaderItem(i,headerItem);
    }
}
```

行表头各列的文字标题由一个 QStringList 对象 headerText 初始化存储，如果只是设置行表头各列的标题，使用下面一行语句即可：

```
ui->tableInfo->setHorizontalHeaderLabels(headerText);
```

如果需要更加具体的格式设置，需要为行表头的每个单元格创建一个 QTableWidgetItem 类型的变量，并进行相应设置。

在一个表格中，不管是表头还是工作区，每个单元格都是一个 QTableWidgetItem 对象。QTableWidgetItem 对象存储了单元格的所有内容，包括字标题、格式设置，以及关联的数据。

上面程序中的 for 循环遍历 headerText 的每一行，用每一行的文字创建一个 QTableWidgetItem 对象

headerItem, 然后设置 headerItem 的字体大小为 12、粗体、红色, 然后将 headerItem 赋给表头的某一列:

```
ui->tableInfo->setHorizontalHeaderItem(i,headerItem);
```

## 2. 初始化表格数据

界面上的“初始化表格数据”按钮根据表格的行数, 生成数据填充表格, 并为每个单元格生成 QTableWidget 对象, 设置相应属性。下面是 btnIniData 的 clicked() 信号的槽函数代码:

```
void MainWindow::on_btnIniData_clicked()
{ //初始化表格内容
    QString strName,strSex;
    bool    isParty=false;
    QDate   birth;
    birth.setDate(1980,4,7); //初始化一个日期
    ui->tableInfo->clearContents(); //只清除工作区, 不清除表头
    int Rows=ui->tableInfo->rowCount(); //数据区行数
    for (int i=0;i<Rows;i++)
    {
        strName=QString::asprintf("学生%d",i);
        if ((i % 2)==0) //分奇数、偶数行设置性别, 及其图标
            strSex="男";
        else
            strSex="女";
        createItemsARow(i, strName, strSex, birth,"汉族",isParty,70);
        birth=birth.addDays(20); //日期加 20 天
        isParty !=isParty;
    }
}
```

QTableWidget::clearContents() 函数清除表格数据区的所有内容, 但是不清除表头。

QTableWidget::rowCount() 函数返回表格数据区的行数。

在 for 循环里为每一行生成需要显示的数据, 然后调用自定义函数 createItemsARow(), 为表格一行的各个单元格生成 QTableWidget 对象。

createItemsARow() 是在窗体类里自定义的函数, 其实现代码如下:

```
void MainWindow::createItemsARow(int rowNo,QString Name,QString Sex,QDate birth,QString
Nation,bool isPM,int score)
{ //为一行的单元格创建 Items
    QTableWidget *item;
    QString str;
    uint StudID=201605000; //学号基数
    //姓名
    item=new QTableWidget(Name,MainWindow::ctName);
    item->setTextAlignment(Qt::AlignHCenter | Qt::AlignVCenter);
    StudID +=rowNo; //学号=基数+ 行号
    item->setData(Qt::UserRole,QVariant(StudID)); //设置 studID 为 data
    ui->tableInfo->setItem(rowNo,MainWindow::colName,item);
    //性别
    QIcon icon;
    if (Sex=="男")
        icon.addFile(":/images/icons/boy.ico");
    else
        icon.addFile(":/images/icons/girl.ico");
```

```

        item=new QTableWidgetItem(Sex,MainWindow::ctSex);
        item->setIcon(icon);
        item->setTextAlignment(Qt::AlignHCenter | Qt::AlignVCenter);
        ui->tableInfo->setItem(rowNo,MainWindow::colSex,item);
//出生日期
        str=birth.toString("yyyy-MM-dd"); //日期转换为字符串
        item=new QTableWidgetItem(str,MainWindow::ctBirth);
        item->setTextAlignment(Qt::AlignLeft | Qt::AlignVCenter);
        ui->tableInfo->setItem(rowNo,MainWindow::colBirth,item);
//民族
        item=new QTableWidgetItem(Nation,MainWindow::ctNation);
        item->setTextAlignment(Qt::AlignHCenter | Qt::AlignVCenter);
        ui->tableInfo->setItem(rowNo,MainWindow::colNation,item);
//是否党员
        item=new QTableWidgetItem("党员",MainWindow::ctPartyM);
        item->setTextAlignment(Qt::AlignHCenter | Qt::AlignVCenter);
        if (isPM)
            item->setCheckState(Qt::Checked);
        else
            item->setCheckState(Qt::Unchecked);
        item->setBackgroundColor(Qt::yellow);
        ui->tableInfo->setItem(rowNo,MainWindow::colPartyM,item);
//分数
        str.setNum(score);
        item=new QTableWidgetItem(str,MainWindow::ctScore);
        item->setTextAlignment(Qt::AlignHCenter | Qt::AlignVCenter);
        ui->tableInfo->setItem(rowNo,MainWindow::colScore,item);
    }

```

该表格的每一行有 5 列，为每一个单元格都创建一个 QTableWidgetItem 类型的变量 item，并做相应的设置。

创建 QTableWidgetItem 使用的构造函数的原型为：

```
QTableWidgetItem::QTableWidgetItem(const QString &text, int type = Type)
```

其中，第一个参数作为单元格的显示文字，第二个参数作为节点的类型。

例如创建“姓名”单元格对象时的语句是：

```
cellItem=new QTableWidgetItem(Name,MainWindow::ctName);
```

其中，MainWindow::ctName 是定义的枚举类型 CellType 的一个常量值。

“姓名”单元格还调用 setData() 函数设置了一个自定义的数据，存储的是学生 ID。

```
cellItem->setData(Qt::UserRole,QVariant(StudID));
```

这个自定义数据是不显示在界面上的，但是与单元格相关联。

QTableWidgetItem 有一些函数对单元格进行属性设置，如下。

- setTextAlignment(int alignment): 设置文字对齐方式。
- setBackground(const QBrush &brush): 设置单元格背景颜色。
- setForeground(const QBrush &brush): 设置单元格前景色。
- setIcon(const QIcon &icon): 为单元格设置一个显示图标。
- setFont(const QFont &font): 为单元格显示文字设置字体。

- `setCheckState(Qt::CheckState state)`: 设置单元格勾选状态, 单元格里出现一个 `QCheckBox` 组件。
- `setFlags(Qt::ItemFlags flags)`: 设置单元格的一些属性标记。

设置好 item 的各种属性之后, 用 `QTableWidget` 的 `setItem` 函数将 item 设置为单元格的项, 例如:

```
ui->tableInfo->setItem(rowNo, MainWindow::colName, item);
```

其中, `MainWindow::colName` 是定义的枚举类型 `FieldColNum` 的一个常量值。

这样初始化设置后, 就可以得到如图 4-18 所示的运行时的表格内容。表格里并没有显示学号, 学号是“姓名”单元格的关联数据。

### 3. 获得当前单元格数据

当鼠标在表格上单击单元格时, 被选中的单元格是当前单元格。通过 `QTableWidget` 的 `currentColumn()` 和 `currentRow()` 可以获得当前单元格的列编号和行编号。

当前单元格发生切换时, 会发射 `currentCellChanged()` 信号和 `currentItemChanged()` 信号, 两个信号都可以利用, 只是传递的参数不同。

对 `currentCellChanged()` 信号编写槽函数, 用于获取当前单元格的数据, 以及当前行的学生的学号信息, 代码如下:

```
void MainWindow::on_tableInfo_currentCellChanged(int currentRow, int currentColumn,
int previousRow, int previousColumn)
{ //当前选择单元格发生变化时的响应
    QTableWidgetItem* item=ui->tableInfo->item(currentRow, currentColumn);
    if (item==NULL)
        return;
    labCellIndex->setText(QString::asprintf("当前单元格坐标: %d 行, %d 列",
        currentRow, currentColumn));
    int cellType=item->type(); //获取单元格的类型
    labCellType->setText(QString::asprintf("当前单元格类型: %d", cellType));
    item=ui->tableInfo->item(currentRow, MainWindow::colName); //第 1 列的 item
    int ID=item->data(Qt::UserRole).toInt(); //读取用户自定义数据
    labStudID->setText(QString::asprintf("学生 ID: %d", ID)); //学生 ID
}
```

在 `currentCellChanged()` 信号中, 传递的参数 `currentRow` 和 `currentColumn` 表示当前单元格的行号和列号, 通过这两个编号可以得到单元格的 `QTableWidgetItem` 对象 `item`。

获得 `item` 之后, 通过 `type()` 函数得到单元格的类型参数, 这个类型就是为单元格创建 `QTableWidgetItem` 对象时传递的类型参数。

再获取同一行的“姓名”单元格的项, 用 `data()` 函数提取自定义数据, 也就是创建单元格时存储的学生 ID。

### 4. 插入、添加、删除行

`QTableWidget` 处理行操作的函数如下。

- `insertRow(int row)`: 在行号为 `row` 的行前面插入一行, 如果 `row` 等于或大于总行数, 则在表格最后添加一行。`insertRow()` 函数只是插入一个空行, 不会为单元格创建 `QTableWidgetItem` 对象, 需要手工为单元格创建。
- `removeRow(int row)`: 删除行号为 `row` 的行。



下面是界面上“插入行”“添加行”“删除当前行”按钮的响应代码。在插入行之后，会调用 `createItemsARow()` 函数，为新创建的空行的各单元格构造 `QTableWidgetItem` 对象。

```
void MainWindow::on_btnInsertRow_clicked()
{ //插入一行
    int curRow=ui->tableInfo->currentRow();
    ui->tableInfo->insertRow(curRow); //插入一行，不会自动为单元格创建 item
    createItemsARow(curRow, "新学生", "男",
        QDate::fromString("1990-1-1", "yyyy-M-d"), "苗族", true );
}
void MainWindow::on_btnAppendRow_clicked()
{ //添加一行
    int curRow=ui->tableInfo->rowCount();
    ui->tableInfo->insertRow(curRow); //在表格尾部添加一行
    createItemsARow(curRow, "新生", "女",
        QDate::fromString("2000-1-1", "yyyy-M-d"), "满族", false );
}
void MainWindow::on_btnDelCurRow_clicked()
{ //删除当前行及其 items
    int curRow=ui->tableInfo->currentRow();
    ui->tableInfo->removeRow(curRow); //删除当前行及其 items
}
```

## 5. 自动调整行高和列宽

`QTableWidget` 有几个函数自动调整表格的行高和列宽，分别如下。

- `resizeColumnsToContents()`: 自动调整所有列的宽度，以适应其内容。
- `resizeColumnToContents(int column)`: 自动调整列号为 `column` 的列的宽度。
- `resizeRowsToContents()`: 自动调整所有行的高度，以适应其内容。
- `resizeRowToContents(int row)`: 自动调整行号为 `row` 的行的高度。

这几个函数实际上是 `QTableWidget` 的父类 `QTableView` 的函数。

## 6. 其他属性控制

- 设置表格内容是否可编辑

`QTableWidget` 的 `EditTriggers` 属性表示是否可编辑，以及进入编辑状态的方式。界面上的“表格可编辑”复选框的槽函数代码为：

```
void MainWindow::on_chkBoxTabEditable_clicked(bool checked)
{ //设置编辑模式
    if (checked)
        ui->tableInfo->setEditTriggers(QAbstractItemView::DoubleClicked |
            QAbstractItemView::SelectedClicked); //双击或获取焦点后单击，进入编辑状态
    else
        ui->tableInfo->setEditTriggers(QAbstractItemView::NoEditTriggers);
}
```

- 设置行表头、列表头是否显示

`horizontalHeader()` 获取行表头，`verticalHeader()` 获取列表头，然后可设置其可见性。

```
void MainWindow::on_chkBoxHeaderH_clicked(bool checked)
{ //是否显示行表头
    ui->tableInfo->horizontalHeader()->setVisible(checked);
}
```

```

}
void MainWindow::on_chkBoxHeaderV_clicked(bool checked)
{ //是否显示列表头
    ui->tableInfo->verticalHeader()->setVisible(checked);
}

```

### • 间隔行底色

setAlternatingRowColors()函数可以设置表格的行是否用交替底色显示，若为交替底色，则间隔的一行会用灰色作为底色。具体底色的设置需要用 styleSheet，在 16.2 节有介绍。

```

void MainWindow::on_chkBoxRowColor_clicked(bool checked)
{
    ui->tableInfo->setAlternatingRowColors(checked);
}

```

### • 选择模式

setSelectionBehavior()函数可以设置选择方式为单元格选择，还是行选择。

```

void MainWindow::on_rBtnSelectItem_clicked()
{ //选择行为：单元格选择
    ui->tableInfo->setSelectionBehavior(QAbstractItemView::SelectItems);
}
void MainWindow::on_rBtnSelectRow_clicked()
{ //选择行为：行选择
    ui->tableInfo->setSelectionBehavior(QAbstractItemView::SelectRows);
}

```

## 7. 遍历表格读取数据

“读取表格内容到文本”按钮演示了将表格数据区的内容全部读出的方法，它将每个单元格的文字读出，同一行的单元格的文字用空格分隔开，作为文本的一行，然后将这行文字作为文本编辑器的一行内容，代码如下：

```

void MainWindow::on_btnReadToEdit_clicked()
{ //将所有单元格的内容提取字符串，显示在 PlainTextEdit 组件里
    QString str;
    QTableWidgetItem *cellItem;
    ui->textEdit->clear();
    for (int i=0; i<ui->tableInfo->rowCount(); i++)
    {
        str=QString::asprintf("第 %d 行: ", i+1);
        for (int j=0; j<ui->tableInfo->columnCount()-1; j++)
        {
            cellItem=ui->tableInfo->item(i, j); //获取单元格的 item
            str=str+cellItem->text()+" "; //字符串连接
        }
        cellItem=ui->tableInfo->item(i, colNoPartyM); //最后一列
        if (cellItem->checkState()==Qt::Checked)
            str=str+"党员";
        else
            str=str+"群众";
        ui->textEdit->appendPlainText(str);
    }
}

```

# Model/View 结构

Model/View（模型/视图）结构是 Qt 中用界面组件显示与编辑数据的一种结构，视图（View）是显示和编辑数据的界面组件，模型（Model）是视图与原始数据之间的接口。Model/View 结构的典型应用是在数据库应用程序中，例如数据库中的一个数据表可以在一个 QTableView 组件中显示和编辑。

主要的视图组件有 QListView、QTreeView 和 QTableView，第 4 章介绍的 QListWidget、QTreeWidget 和 QTableWidget 分别是这 3 个类的便利类，它们不使用数据模型，而是将数据直接存储在组件的每个项里。

本章介绍 Model/View 结构原理，包括 QListView、QTreeView、QTableView 视图组件，以及 QStringListModel、StandardItemModel 等模型类的用法。

## 5.1 Model/View 结构

### 5.1.1 Model/View 基本原理

GUI 应用程序的一个很重要的功能是由用户在界面上编辑和修改数据，典型的如数据库应用程序。数据库应用程序中，用户在界面上执行各种操作，实际上是修改了界面组件所关联的数据库内的数据。

将界面组件与所编辑的数据分离开来，又通过数据源的方式连接起来，是处理界面与数据的一种较好的方式。Qt 使用 Model/View 结构来处理这种关系，Model/View 的基本结构如图 5-1 所示。其中各部分的功能如下。

- 数据（Data）是实际的数据，如数据库的一个数据表或 SQL 查询结果，内存中的一个 QStringList，或磁盘文件结构等。
- 视图或视图组件（View）是屏幕上的界面组件，视图从数据模型获得每个数据项的模型索引（model index），通过模型索引获取数据，然后为界面组件提供显示数据。Qt 提供一些现成的数据视图组件，如 QListView、QTreeView 和 QTableView 等。
- 模型或数据模型（Model）与实际数据通信，并为视图组件提供数据接口。它从原始数据

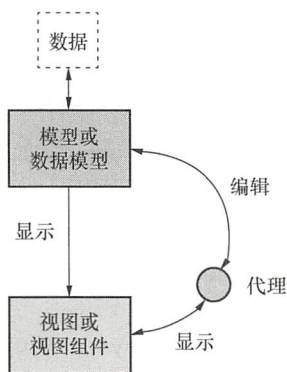


图 5-1 Model/View 基本结构  
(来自 Qt 帮助文件)

提取需要的内容，用于视图组件进行显示和编辑。Qt 中有一些预定义的数据模型，如 QStringListModel 可作为 QStringList 的数据模型，QSqlTableModel 可以作为数据库中一个数据表的数据模型。

由于数据源与显示界面通过 Model/View 结构分离开来，因此可以将一个数据模型在不同的视图中显示，也可以在不修改数据模型的情况下，设计特殊的视图组件。

在 Model/View 结构中，还提供了代理（Delegate）功能，代理功能可以让用户定制数据的界面显示和编辑方式。在标准的视图组件中，代理功能显示一个数据，当数据被编辑时，代理通过模型索引与数据模型通信，并为编辑数据提供一个编辑器，一般是一个 QLineEdit 组件。

模型、视图和代理之间使用信号和槽通信。当源数据发生变化时，数据模型发射信号通知视图组件；当用户在界面上操作数据时，视图组件发射信号表示这些操作信息；当编辑数据时，代理发射信号告知数据模型和视图组件编辑器的状态。

5.1.2 数据模型

所有的基于项数据（item data）的数据模型（Model）都是基于 QAbstractItemModel 类的，这个类定义了视图组件和代理存取数据的接口。数据无需存储在数据模型里，数据可以是其他类、文件、数据库或任何数据源。Qt 中与数据模型相关的几个主要的类的层次结构如图 5-2 所示。

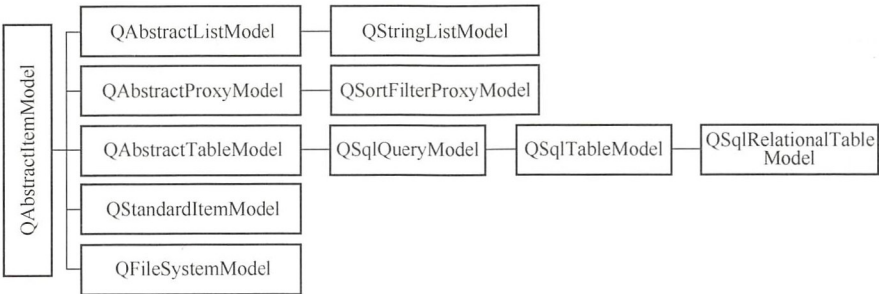


图 5-2 Qt 中模型类的层次结构

图 5-2 中的抽象类是不能直接使用的，需要由子类继承来实现一些纯虚函数。Qt 提供了一些模型类用于项数据处理，常见的几个见表 5-1。

表 5-1 Qt 提供的数据库模型类

Model 类	用途
QStringListModel	用于处理字符串列表数据的数据模型类
QStandardItemModel	标准的基于项数据的数据模型类，每个项数据可以是任何数据类型
QFileSystemModel	计算机上文件系统的数据模型类
QSortFilterProxyModel	与其他数据模型结合，提供排序和过滤功能的数据模型类
QSqlQueryModel	用于数据库 SQL 查询结果的数据模型类
QSqlTableModel	用于数据库的一个数据表的数据模型类
QSqlRelationalTableModel	用于关系型数据表的数据模型类

数据库相关的 3 个模型类将在第 11 章介绍数据库编程时专门说明。如果这些现有的模型类无



法满足需求，用户可以从 `QAbstractItemModel`、`QAbstractListModel` 或 `QAbstractTableModel` 继承，生成自己定制的数据模型类。

### 5.1.3 视图组件

视图组件（View）就是显示数据模型的数据的界面组件，Qt 提供的视图组件如下。

- `QListView`：用于显示单列的列表数据，适用于一维数据的操作。
- `QTreeView`：用于显示树状结构数据，适用于树状结构数据的操作。
- `QTableView`：用于显示表格状数据，适用于二维表格型数据的操作。
- `QColumnView`：用多个 `QListView` 显示树状层次结构，树状结构的一层用一个 `QListView` 显示。
- `QHeaderView`：提供行表头或列表头的视图组件，如 `QTableView` 的行表头和列表头。

视图组件在显示数据时，只需调用视图类的 `setModel()` 函数，为视图组件设置一个数据模型就可以实现视图组件与数据模型之间的关联，在视图组件上的修改将自动保存到关联的数据模型里，一个数据模型可以同时多个视图组件里显示数据。

在第4章介绍了 `QListWidget`、`QTreeWidget` 和 `QTableWidget` 3 个可用于数据编辑的组件。这 3 个类称为便利类（convenience classes），它们分别是 3 个视图类的子类，其层次关系如图 5-3 所示。

用于 Model/View 结构的几个视图类直接从 `QAbstractItemView` 继承而来，而便利类则从相应的视图类继承而来。

视图组件类的数据采用单独的数据模型，视图组件不存储数据。便利类则为组件的每个节点或单元格创建一个项（item），用项存储数据、格式设置等。所以，便利类没有数据模型，它实际上是用项的方式集成了数据模型的功能，这样就将界面与数据绑定了。

所以，便利类缺乏对大型数据源进行灵活处理的能力，适用于小型数据的显示和编辑。

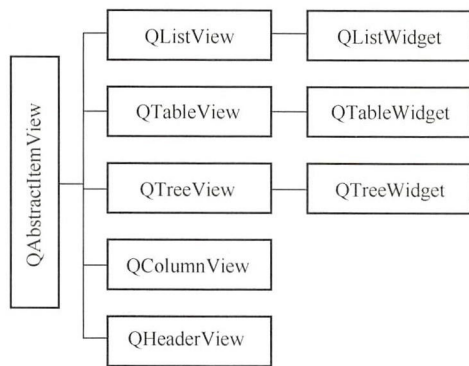


图 5-3 视图相关类的层次结构图

### 5.1.4 代理

代理（Delegate）就是在视图组件上为编辑数据提供编辑器，如在表格组件中编辑一个单元格的数据时，缺省是使用一个 `QLineEdit` 编辑框。代理负责从数据模型获取相应的数据，然后显示在编辑器里，修改数据后，又将其保存到数据模型中。

`QAbstractItemDelegate` 是所有代理类的基类，作为抽象类，它不能直接使用。它的一个子类 `QStyledItemDelegate`，是 Qt 的视图组件缺省使用的代理类。

对于一些特殊的数据编辑需求，例如只允许输入整型数，使用一个 `QSpinBox` 作为代理组件更恰当，从列表中选择数据时使用一个 `QComboBox` 作为代理组件更好。这时，就可以从

QStyledItemDelegate 继承创建自定义代理类。

### 5.1.5 Model/View 结构的一些概念

#### 1. Model/View 的基本结构

在 Model/View 结构中，数据模型为视图组件和代理提供存取数据的标准接口。在 Qt 中，所有的数据模型类都从 QAbstractItemModel 继承而来，不管底层的数据结构是如何组织数据的，QAbstractItemModel 的子类都以表格的层次结构表示数据，视图组件通过这种规则来存取模型中的数据，但是表现给用户的形式不一样。

图 5-4 是数据模型的 3 种常见表现形式。不管数据模型的表现形式是怎么样的，数据模型中存储数据的基本单元都是项 (item)，每个项有一个行号、一个列号，还有一个父项 (parent item)。在列表和表格模式下，所有的项都有一个相同的顶层项 (root item)；在树状结构中，行号、列号、父项稍微复杂一点，但是由这 3 个参数完全可以定义一个项的位置，从而存取项的数据。

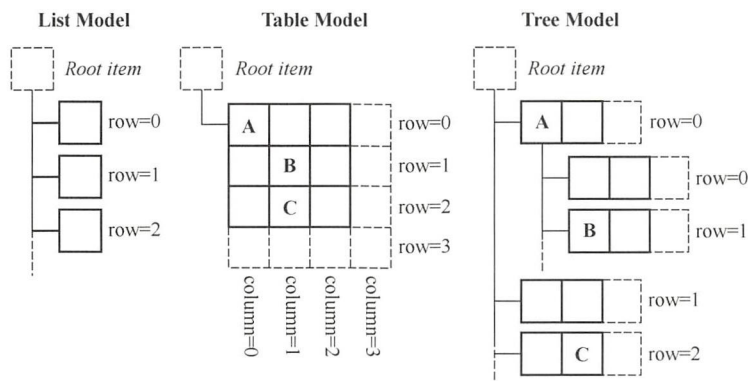


图 5-4 数据模型的几种表现形式 (来自 Qt 帮助文件)

#### 2. 模型索引

为了保证数据的表示与数据存取方式隔离，数据模型中引入了模型索引 (model index) 的概念。通过数据模型存取的每个数据都有一个模型索引，视图组件和代理都通过模型索引来获取数据。

QModelIndex 表示模型索引的类。模型索引提供数据存取的一个临时指针，用于通过数据模型提取或修改数据。因为模型内部组织数据的结构随时可能改变，所以模型索引是临时的。如果需要使用持久性的模型索引，则要使用 QPersistentModelIndex 类。

#### 3. 行号和列号

数据模型的基本形式是用行和列定义的表格数据，但这并不意味着底层的数据是用二维数组存储的，使用行和列只是为了组件之间交互方便的一种规定。通过模型索引的行号和列号就可以存取数据。

要获得一个模型索引，必须提供 3 个参数：行号、列号、父项的模型索引。例如，对于如图 5-4 中的表格数据模型中的 3 个数据项 A、B、C，获取其模型索引的代码是：

```
QModelIndex indexA = model->index(0, 0, QModelIndex());
QModelIndex indexB = model->index(1, 1, QModelIndex());
QModelIndex indexC = model->index(2, 1, QModelIndex());
```

在创建模型索引的函数中需要传递行号、列号和父项的模型索引。对于列表和表格模式的数据模型，顶层节点总是用 `QModelIndex()` 表示。

#### 4. 父项

当数据模型是列表或表格时，使用行号、列号存储数据比较直观，所有数据项的父项（parent item）就是顶层项；当数据模型是树状结构时，情况比较复杂（树状结构中，项一般习惯于称为节点），一个节点可以有父节点，也可以是其他节点的父节点，在构造数据项的模型索引时，必须指定正确的行号、列号和父节点。

对于图 5-4 中的树状数据模型，节点 A 和节点 C 的父节点是顶层节点，获取模型索引的代码是：

```
QModelIndex indexA = model->index(0, 0, QModelIndex());
QModelIndex indexC = model->index(2, 1, QModelIndex());
```

但是，节点 B 的父节点是节点 A，节点 B 的模型索引由下面的代码生成：

```
QModelIndex indexB = model->index(1, 0, indexA);
```

#### 5. 项的角色

在为数据模型的一个项设置数据时，可以赋予其不同项的角色（item role）的数据。例如，数据模型类 `QStandardItemModel` 的项数据类是 `QStandardItem`，其设置数据的函数是：

```
void QStandardItem::setData(const QVariant &value, int role = Qt::UserRole + 1)
```

其中，`value` 是需要设置的数据，`role` 是设置数据的角色。一个项可以有不同角色的数据，用于不同的场合。

`role` 是 `Qt::ItemDataRole` 枚举类型，有多种取值，如 `Qt::DisplayRole` 角色是在视图组件中显示的字符串，`Qt::ToolTipRole` 是鼠标提示消息，`Qt::UserRole` 可以自定义数据。项的标准角色是 `Qt::DisplayRole`。

在获取一个项的数据时也需要指定角色，以获取不同角色的数据。

```
QVariant QStandardItem::data(int role = Qt::UserRole + 1) const
```

为一个项的不同角色定义数据，可以告知视图组件和代理组件如何显示数据。例如，在图 5-5 中，项的 `DisplayRole` 数据是显示的字符串，`DecorationRole` 是用于装饰显示的属性，`ToolTipRole` 定义了鼠标提示信息。不同的视图组件对各种角色数据的解释和显示可能不一样，也可能忽略某些角色的数据。

第 4 章已经介绍了便利类 `QListWidget`、`QTreeWidget` 和 `QTableWidget` 的使用，本章将介绍 Model/View 结构的基本用法，包括 Qt 预定义的 `QStringListModel`、`QFileSystemModel`、`QStandardItemModel` 以及视图组件 `QListView`、`QTableView`、`QTreeView` 的

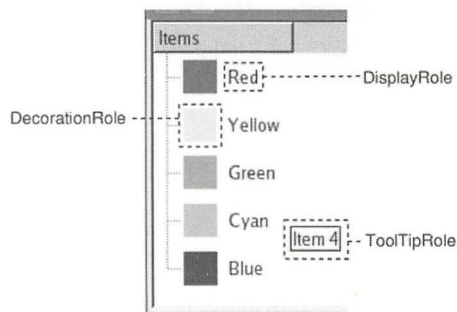


图 5-5 不同角色数据的表现形式（来自 Qt 帮助文件）



使用, 还介绍如何设计和使用自定义代理。涉及数据库的 Model/View 的使用将在数据库一章单独介绍。

## 5.2 QFileSystemModel

### 5.2.1 QFileSystemModel 类的基本功能

QFileSystemModel 提供了一个可用于访问本机文件系统的数据模型。QFileSystemModel 和视图组件 QTreeView 结合使用, 可以用目录树的形式显示本机上的文件系统, 如同 Windows 的资源管理器一样。使用 QFileSystemModel 提供的接口函数, 可以创建目录、删除目录、重命名目录, 可以获得文件名称、目录名称、文件大小等参数, 还可以获得文件的详细信息。

要通过 QFileSystemModel 获得本机的文件系统, 需要用 setRootPath() 函数为 QFileSystemModel 设置一个根目录, 例如:

```
QFileSystemModel *model = new QFileSystemModel;  
model->setRootPath(QDir::currentPath());
```

静态函数 QDir::currentPath() 获取应用程序的当前路径。

用于获取磁盘文件目录的数据模型类还有一个 QDirModel, QDirModel 的功能与 QFileSystemModel 类似, 也可以获取目录和文件, 但是 QFileSystemModel 采用单独的线程获取目录文件结构, 而 QDirModel 不使用单独的线程。使用单独的线程就不会阻碍主线程, 所以推荐使用 QFileSystemModel。

使用 QFileSystemModel 作为数据模型, QTreeView、QListView 和 QTableView 为主要组件设计的实例 samp5\_1 运行界面如图 5-6 所示。在 TreeView 中以目录树的形式显示本机的文件系统, 单击一个目录时, 右边的 ListView 和 TableView 显示该目录下的目录和文件。在 TreeView 上单击一个目录或文件节点时, 下方的几个标签里显示当前节点的信息。

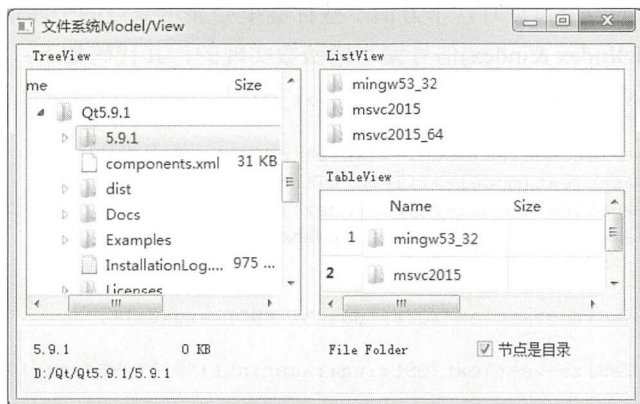


图 5-6 实例 samp5\_1 的运行时界面

### 5.2.2 QFileSystemModel 的使用

实例 samp5\_1 的主窗口是基于 QMainWindow 的, 在使用 UI 设计器做可视化设计时删除了工



具栏和状态栏。主窗口界面布局采用了两个分割条的设计，ListView 和 TableView 采用上下分割布局，然后和左边的 TreeView 采用水平分割布局，水平分割布局再和下方显示信息的 groupBox 在主窗口工作区水平布局。

在主窗口类中定义了一个 QFileSystemModel 类的成员变量 model。

```
QFileSystemModel *model;
```

主窗口构造函数进行初始化，代码如下：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    model=new QFileSystemModel(this);
    model->setRootPath(QDir::currentPath()); //设置根目录
    ui->treeView->setModel(model); //设置数据模型
    ui->listView->setModel(model); //设置数据模型
    ui->tableView->setModel(model); //设置数据模型
    //信号与槽关联，treeView 单击时，其目录设置为 listView 和 tableView 的根节点
    connect(ui->treeView, SIGNAL(clicked(QModelIndex)),
            ui->listView, SLOT(setRootIndex(QModelIndex)));
    connect(ui->treeView, SIGNAL(clicked(QModelIndex)),
            ui->tableView, SLOT(setRootIndex(QModelIndex)));
}
```

3 个视图组件都使用 setModel()函数，将 QFileSystemModel 数据模型 model 设置为自己的数据模型。

connect()函数设置信号与槽的关联，实现的功能是：在单击 treeView 的一个节点时，此节点就设置为 listView 和 tableView 的根节点，因为 treeView 的 clicked(QModelIndex)信号会传递一个 QModelIndex 变量，是当前节点的模型索引，将此模型索引传递给 listView 和 tableView 的槽函数 setRootIndex(QModelIndex)，listView 和 tableView 就会显示此节点下的目录和文件。

在 treeView 上单击一个节点时，下方的一些标签里会显示节点的一些信息，这是为 treeView 的 clicked(const QModelIndex &index)信号编写槽函数实现的，其代码如下：

```
void MainWindow::on_treeView_clicked(const QModelIndex &index)
{
    ui->chkIsDir->setChecked(model->isDir(index)); //是否是目录
    ui->LabPath->setText(model->filePath(index));
    ui->LabType->setText(model->type(index));
    ui->LabFileName->setText(model->fileName(index));
    int sz=model->size(index)/1024;
    if (sz<1024)
        ui->LabFileSize->setText(QString("%1 KB").arg(sz));
    else
        ui->LabFileSize->setText(QString::asprintf("%.1f MB",sz/1024.0));
}
```

函数有一个传递参数 QModelIndex &index，它是单击节点在数据模型中的索引。通过传递来的模型索引 index，这段代码使用了 QFileSystemModel 的一些函数来获得节点的一些参数，包括以下几种。

- bool isDir(QModelIndex &index)：判断节点是不是一个目录。
- QString filePath(QModelIndex &index)：返回节点的目录名或带路径的文件名。

- QString fileName(QModelIndex &index): 返回去除路径的文件夹名称或文件名。
- QString type(QModelIndex &index): 返回描述节点类型的文字, 如硬盘符是“Drive”, 文件夹是“File Folder”, 文件则用具体的后缀描述, 如“txt File”“exe File”“pdf File”等。
- qint64 size(QModelIndex &index): 如果节点是文件, 返回文件大小的字节数; 如果节点是文件夹, 返回 0。

而 QFileSystemModel 是如何获取磁盘目录文件结构的, 3 个视图组件是如何显示这些数据的, 则是其底层实现的问题了。

## 5.3 QStringListModel

### 5.3.1 QStringListModel 功能概述

QStringListModel 用于处理字符串列表的数据模型, 它可以作为 QListView 的数据模型, 在界面上显示和编辑字符串列表。

QStringListModel 的 setStringList() 函数可以初始化数据模型的字符串列表的内容, stringList() 函数返回数据模型内的字符串列表, 在关联的 ListView 组件里编辑修改数据后, 数据都会及时更新到数据模型内的字符串列表里。

QStringListModel 提供编辑和修改字符串列表数据的函数, 如 insertRows()、removeRows()、setData() 等, 这些操作直接影响数据模型内部的字符串列表, 并且修改后的数据会自动在关联的 ListView 组件里刷新显示。

实例 samp5\_2 采用 QStringListModel 作为数据模型, QListView 组件作为视图组件, 演示了 QStringListModel 和 QListView 构成 Model/View 结构编辑字符串列表的功能, 程序运行时界面如图 5-7 所示。

窗口左侧是对 QStringListModel 的一些操作, 右侧的 QPlainTextEdit 组件显示 QStringListModel::stringList() 的内容, 以查看其是否与界面上 ListView 组件显示的内容一致。

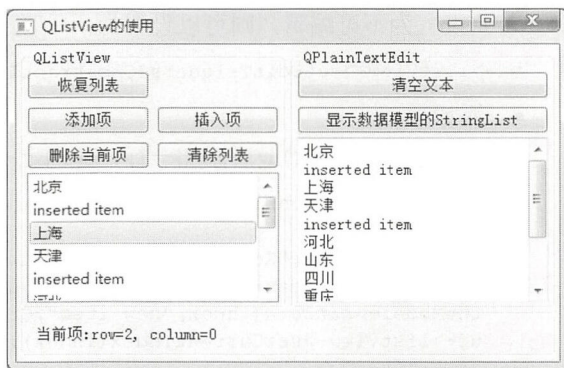


图 5-7 实例 samp5\_2 的运行时界面

### 5.3.2 QStringListModel 的使用

#### 1. Model/View 结构对象和组件初始化

实例 samp5\_2 的窗口是从 QWidget 继承而来的类 Widget, 界面采用可视化设计。在 Widget 类中定义一个 QStringListModel 类的变量:

```
QStringListModel *theModel;
```

在 Widget 类的构造函数中进行变量的创建, 完成数据模型与界面视图组件的关联, 下面是

Widget 类构造函数的代码:

```
Widget::Widget(QWidget *parent) :    QWidget(parent),    ui(new Ui::Widget)
{
    ui->setupUi(this);
    QStringList theStrList;
    theStrList<<"北京"<<"上海"<<"天津"<<"河北"<<"山东"<<"四川"<<"重庆";
    theModel=new QStringListModel(this);
    theModel->setStringList(theStrList); //导入 theStrList 的内容
    ui->listView->setModel(theModel); //设置数据模型
    ui->listView->setEditTriggers(QAbstractItemView::DoubleClicked |
                                QAbstractItemView::SelectedClicked);
}
```

QStringListModel 的 setStringList()函数将一个字符串列表的内容作为数据模型的初始数据内容。

QListView 的 setModel()函数为界面视图组件设置一个数据模型。

程序运行后,界面上 ListView 组件里就会显示初始化的字符串列表的内容。

## 2. 编辑、添加、删除项的操作

### • 编辑项

QListView::setEditTriggers()函数设置 QListView 的条目是否可以编辑,以及如何进入编辑状态,函数的参数是 QAbstractItemView::EditTrigger 枚举类型值的组合。构造函数中设置为:

```
ui->listView->setEditTriggers(QAbstractItemView::DoubleClicked |
                              QAbstractItemView::SelectedClicked);
```

表示在双击,或选择并单击列表项后,就进入编辑状态。

若要设置为不可编辑,则可以设置为:

```
ui->listView->setEditTriggers(QAbstractItemView::NoEditTriggers);
```

### • 添加项

添加项是要在列表的最后添加一行,界面上“添加项”按钮的槽函数代码如下:

```
void Widget::on_btnListAppend_clicked()
{ //添加一行
    theModel->insertRow(theModel->rowCount()); //在尾部插入一空行
    QModelIndex index=theModel->index(theModel->rowCount()-1,0);
    theModel->setData(index,"new item",Qt::DisplayRole);
    ui->listView->setCurrentIndex(index); //设置当前选中的行
}
```

对数据的操作都是针对数据模型的,所以,插入一行使用的是 QStringListModel 的 insertRow (int row)函数,其中 row 是一个行号,表示在 row 行之前插入一行。要在列表的最后插入一行,参数 row 设置为列表当前的行数即可。

这样只是在列表尾部添加一个空行,没有任何文字。为了给添加的项设置一个缺省的文字标题,首先要获得新增项的模型索引,即:

```
QModelIndex index=theModel->index(theModel->rowCount()-1,0);
```

QStringListModel 的 index()函数根据传递的行号、列号、父项的模型索引生成一个模型索引,这行代码为新增的最后一个项生成一个模型索引 index。



为新增的项设置一个文字标题“new item”，使用 `setData()` 函数，并用到前面生成的模型索引 `index`。代码如下：

```
theModel->setData(index, "new item", Qt::DisplayRole);
```

在使用 `setData()` 函数时，必须指定设置数据的角色，这里的角色是 `Qt::DisplayRole`，它是用于显示的角色，即项的文字标题。

- 插入项

“插入项”按钮的功能是在列表的当前行前面插入一行，其实现代码如下：

```
void Widget::on_btnListInsert_clicked()
{ // 插入一行
    QModelIndex index=ui->listView->currentIndex();
    theModel->insertRow(index.row());
    theModel->setData(index, "inserted item", Qt::DisplayRole);
    ui->listView->setCurrentIndex(index);
}
```

`QListView::currentIndex()` 获得当前项的模型索引 `index`，`index.row()` 则返回这个模型索引的行号。

- 删除当前项

使用 `QStringListModel` 的 `removeRow()` 函数删除某一行的代码如下：

```
void Widget::on_btnListDelete_clicked()
{ // 删除当前行
    QModelIndex index=ui->listView->currentIndex();
    theModel->removeRow(index.row());
}
```

- 删除列表

删除列表的所有项可使用 `QStringListModel` 的 `removeRows(int row, int count)` 函数，它表示从行号 `row` 开始删除 `count` 行。代码如下：

```
void Widget::on_btnListClear_clicked()
{ // 清除所有项
    theModel->removeRows(0, theModel->rowCount());
}
```

### 3. 以文本显示数据模型的内容

以上在对界面上 `ListView` 的项进行编辑时，实际操作的都是其关联的数据模型 `theModel`，在对数据模型进行插入、添加、删除项操作后，内容立即在 `ListView` 上显示出来，这是数据模型与视图组件之间信号与槽的作用，当数据模型的内容发生改变时，通知视图组件更新显示。

同样的，当在 `ListView` 上双击一行进入编辑状态，修改一个项的文字内容后，这部分内容也保存到数据模型里了，这就是图 5-1 所表示的过程。

那么，数据模型内部应该保存有最新的数据内容，对于 `QStringListModel` 模型来说，通过 `stringList()` 函数可以得到其最新的数据副本。界面中的“显示数据模型的 `StringList`”按钮获取数据模型的 `stringList`，并用多行文本的形式显示其内容，以检验对数据模型修改数据，特别是在界面上修改列表项的文字后，其内部的数据是否同步更新了。

以下是界面上的“显示数据模型的 `StringList`”按钮的 `clicked()` 信号的槽函数代码，它通过数



据模型的 `stringList()` 函数获取字符串列表，并在 `plainTextEdit` 里逐行显示：

```
void Widget::on_btnTextImport_clicked()
{ //显示数据模型的 QStringList
    QStringList tmpList=theModel->stringList();
    ui->plainTextEdit->clear();
    for (int i=0; i<tmpList.count();i++)
        ui->plainTextEdit->appendPlainText(tmpList.at(i));
}
```

程序运行时，无论对 `ListView` 的列表做了什么编辑和修改，单击“显示数据模型的 `StringList`”按钮，在文本框里显示的文字内容与 `ListView` 里总是完全相同的，说明数据模型的数据与界面上显示的内容是同步的。

#### 4. 其他功能

`QListView` 的 `clicked()` 信号会传递一个 `QModelIndex` 类型的参数，利用该参数，可以显示当前项的模型索引的行和列的信息，实现代码如下：

```
void Widget::on_listView_clicked(const QModelIndex &index)
{ //显示 QModelIndex 的行、列号
    ui->LabInfo->setText(QString::asprintf("当前条目:row=%d, column=%d",
        index.row(),index.column()));
}
```

在这个实例中，通过 `QStringListModel` 和 `QListView` 说明了数据模型与视图组件之间构成 `Model/View` 结构的基本原理。

第4章的实例 `samp4_7` 中采用 `QListWidget` 设计了一个列表编辑器，对比这两个实例，可以发现如下两点。

- 在 `Model/View` 结构中，数据模型与视图组件是分离的，可以直接操作数据模型以修改数据，在视图组件中做的修改也会自动保存到数据模型里。
- 在使用 `QListWidget` 的例子中，每个列表项是一个 `QListWidgetItem` 类型的变量，保存了项的各种数据，数据和显示界面是一体的，对数据的修改操作就是对项关联的变量的修改。所以，这是 `Model/View` 结构与便利组件之间的主要区别。

## 5.4 QStandardItemModel

### 5.4.1 功能概述

`QStandardItemModel` 是标准的以项数据（item data）为基础的标准数据模型类，通常与 `QTableView` 组合成 `Model/View` 结构，实现通用的二维数据的管理功能。

本节介绍 `QStandardItemModel` 的使用，主要用到以下3个类。

- `QStandardItemModel`：基于项数据的标准数据模型，可以处理二维数据。维护一个二维的项数据数组，每个项是一个 `QStandardItem` 类的变量，用于存储项的数据、字体格式、对齐方式等。
- `QTableView`：二维数据表视图组件，有多个行和多个列，每个基本显示单元是一个单元格，通过 `setModel()` 函数设置一个 `QStandardItemModel` 类的数据模型之后，一个单元格显示

QStandardItemModel 数据模型中的一个项。

- QItemSelectionModel: 一个用于跟踪视图组件的单元格选择状态的类, 当在 QTableView 选择某个单元格, 或多个单元格时, 通过 QItemSelectionModel 可以获得选中的单元格的模型索引, 为单元格的选择操作提供方便。

这几个类之间的关系是: QTableView 是界面视图组件, 其关联的数据模型是 QStandardItemModel, 关联的项选择模型是 QItemSelectionModel, QStandardItemModel 的数据管理的基本单元是 QStandardItem。

实例 samp5\_3 演示 QStandardItemModel 的使用, 其运行时界面如图 5-8 所示。

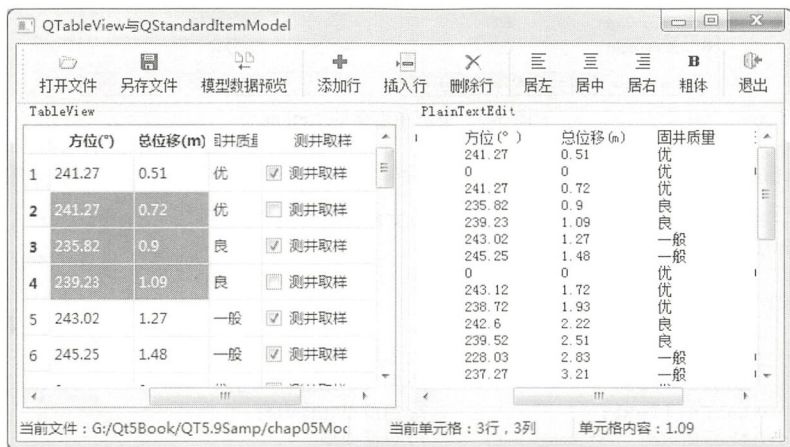


图 5-8 实例 samp5\_3 的运行界面

该实例具有如下功能。

- 打开一个纯文本文件, 该文件是规则的二维数据文件, 通过字符串处理获取表头和各行各列的数据, 导入到一个 QStandardItemModel 数据模型。
- 编辑修改数据模型的数据, 可以插入行、添加行、删除行, 还可以在 QTableView 视图组件中直接修改单元格的数据内容。
- 可以设置数据模型中某个项的不同角色的数据, 包括文字对齐方式、字体是否粗体等。
- 通过 QItemSelectionModel 获取视图组件上的当前单元格, 以及选择单元格的范围, 对选择的单元格进行操作。
- 将数据模型的数据内容显示到 QPlainTextEdit 组件里, 显示数据模型的内容, 检验视图组件上做的修改是否与数据模型同步。
- 将修改后的模型数据另存为一个文本文件。

## 5.4.2 界面设计与主窗口类定义

本实例的主窗口从 QMainWindow 继承而来, 中间的 TableView 和 PlainTextEdit 组件采用水平分割条布局。在 Action 编辑器中创建如图 5-9 所示的一些 Action, 并由 Action 创建主工具栏上的

按钮，下方的状态栏设置了几个 QLabel 组件，显示当前文件名称、当前单元格行号、列号，以及相应内容。

Name	Used	Text	Shortcut	Checkable	ToolTip
 actOpen	<input checked="" type="checkbox"/>	打开文件		<input type="checkbox"/>	打开文件
 actSave	<input checked="" type="checkbox"/>	另存文件		<input type="checkbox"/>	表格内容另存为文件
 actAppend	<input checked="" type="checkbox"/>	添加行		<input type="checkbox"/>	添加一行
 actInsert	<input checked="" type="checkbox"/>	插入行		<input type="checkbox"/>	插入一行
 actDelete	<input checked="" type="checkbox"/>	删除行		<input type="checkbox"/>	删除当前行
 actExit	<input checked="" type="checkbox"/>	退出		<input type="checkbox"/>	退出
 actModelData	<input checked="" type="checkbox"/>	模型数据预览		<input type="checkbox"/>	模型数据显示到文本框里
 actAlignLeft	<input checked="" type="checkbox"/>	居左		<input type="checkbox"/>	文字左对齐
 actAlignCenter	<input checked="" type="checkbox"/>	居中		<input type="checkbox"/>	文字居中
 actAlignRight	<input checked="" type="checkbox"/>	居右		<input type="checkbox"/>	文字右对齐
 actFontBold	<input checked="" type="checkbox"/>	粗体		<input checked="" type="checkbox"/>	粗体字体
Action Editor    Signals & Slots Editor					

图 5-9 实例中创建的 Action

主窗口类 MainWindow 里新增的定义如下（省略了 UI 设计器生成的界面组件的槽函数的声明）：

```
#define FixedColumnCount 6 //文件固定 6 列
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QLabel *LabCurFile; //当前文件
    QLabel *LabCellPos; //当前单元格行列号
    QLabel *LabCellText; //当前单元格内容
    QStandardItemModel *theModel; //数据模型
    QItemSelectionModel *theSelection; //选择模型
    void iniModelFromStringList(QStringList&); //从 QStringList 初始化数据模型
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    //当前选择单元格发生变化
    void on_currentChanged(const QModelIndex &current, const QModelIndex &previous);
private:
    Ui::MainWindow *ui;
};
```

这里定义了数据模型变量 theModel，项数据选择模型变量 theSelection。

定义的私有函数 iniModelFromStringList() 用于在打开文件时，从一个 QStringList 变量的内容创建数据模型。

自定义槽函数 on\_currentChanged() 用于在 TableView 上选择单元格发生变化时，更新状态栏的信息显示，这个槽函数将会与项选择模型 theSelection 的 currentChanged() 信号关联。

### 5.4.3 QStandardItemModel 的使用

#### 1. 系统初始化

在 MainWindow 的构造函数中进行界面初始化，数据模型和选择模型的创建，以及与视图组



件的关联，信号与槽的关联等设置，代码如下：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    setCentralWidget(ui->splitter);
    theModel = new QStandardItemModel(2, FixedColumnCount, this); //数据模型
    theSelection = new QItemSelectionModel(theModel); //选择模型
    connect(theSelection, SIGNAL(currentChanged(QModelIndex, QModelIndex)),
            this, SLOT(on_currentChanged(QModelIndex, QModelIndex)));
    ui->tableView->setModel(theModel); //设置数据模型
    ui->tableView->setSelectionModel(theSelection); //设置选择模型
    ui->tableView->setSelectionMode(QAbstractItemView::ExtendedSelection);
    ui->tableView->setSelectionBehavior(QAbstractItemView::SelectItems);
    //创建状态栏组件，代码略
}
```

在构造函数里首先创建数据模型 `theModel`，创建数据选择模型时需要传递一个数据模型变量作为其参数。这样，数据选择模型 `theSelection` 就与数据模型 `theModel` 关联，用于表示 `theModel` 的项数据选择操作。

创建数据模型和选择模型后，为 `TableView` 组件设置数据模型和选择模型：

```
ui->tableView->setModel(theModel); //设置数据模型
ui->tableView->setSelectionModel(theSelection); //设置选择模型
```

构造函数里还将自定义的槽函数 `on_currentChanged()` 与 `theSelection` 的 `currentChanged()` 信号关联，用于界面上 `tableView` 选择单元格发生变化时，显示单元格的行号、列号、内容等信息，槽函数代码如下：

```
void MainWindow::on_currentChanged(const QModelIndex &current, const QModelIndex
&previous)
{ //选择单元格变化时的响应
    if (current.isValid())
    {
        LabCellPos->setText(QString::asprintf("当前单元格: %d 行, %d 列",
                                                current.row(), current.column()));
        QStandardItem* aItem=theModel->itemFromIndex(current);
        this->LabCellText->setText("单元格内容: "+aItem->text());
        QFont font=aItem->font();
        ui->actFontBold->setChecked(font.bold());
    }
}
```

## 2. 从文本文件导入数据

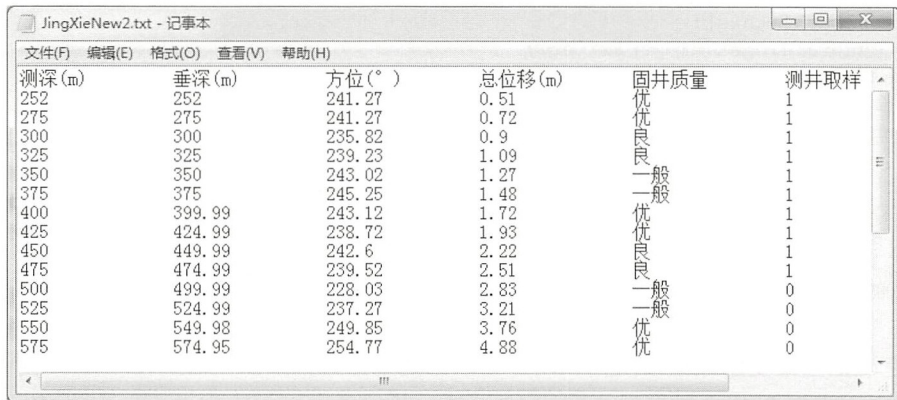
`QStandardItemModel` 是标准的基于项数据的数据模型，以类似于二维数组的形式管理内部数据，适合于处理表格型数据，其显示一般采用 `QTableView`。

`QStandardItemModel` 的数据可以是程序生成的内存中的数据，也可以来源于文件。例如，在实际数据处理中，有些数据经常是以纯文本格式保存的，它们有固定的列数，每一列是一项数据，实际构成一个二维数据表。图 5-10 是本实例程序要打开的一个纯文本文件的内容，文件的第 1 行是数据列的文字标题，相当于数据表的表头，然后以行存储数据，以 TAB 键间隔每列数据。

当单击工具栏上的“打开文件”按钮时，需要选择一个这样的文件导入到数据模型，并在



tableView 上进行显示和编辑。图 5-10 的数据有 6 列，第 1 列是整数，第 2 至 4 列是浮点数，第 5 列是文字，第 6 列是逻辑型变量，“1”表示 true。



测深(m)	垂深(m)	方位(°)	总位移(m)	固井质量	测井取样
252	252	241.27	0.51	优	1
275	275	241.27	0.72	优	1
300	300	235.82	0.9	良	1
325	325	239.23	1.09	良	1
350	350	243.02	1.27	一般	1
375	375	245.25	1.48	一般	1
400	399.99	243.12	1.72	优	1
425	424.99	238.72	1.93	优	1
450	449.99	242.6	2.22	良	1
475	474.99	239.52	2.51	良	1
500	499.99	228.03	2.83	一般	0
525	524.99	237.27	3.21	一般	0
550	549.98	249.85	3.76	优	0
575	574.95	254.77	4.88	优	0

图 5-10 纯文本格式的数据文件

下面是“打开文件”按钮的槽函数代码：

```
void MainWindow::on_actOpen_triggered()
{ //打开文件
    QString curPath=QCoreApplication::applicationDirPath();
    QString aFileName=QFileDialog::getOpenFileName(this, "打开一个文件",
        curPath, "井数据文件 (*.txt);;所有文件 (*.*)");
    if (aFileName.isEmpty())
        return;

    QStringList fFileContent;
    QFile aFile(aFileName);
    if (aFile.open(QIODevice::ReadOnly | QIODevice::Text)) //打开文件
    {
        QTextStream aStream(&aFile); //用文本流读取文件
        ui->plainTextEdit->clear();
        while (!aStream.atEnd())
        {
            QString str=aStream.readLine();
            ui->plainTextEdit->appendPlainText(str);
            fFileContent.append(str);
        }
        aFile.close();
        this->LabCurFile->setText("当前文件: "+aFileName); //状态栏显示
        iniModelFromStringList(fFileContent); //初始化数据模型
    }
}
```

这段代码让用户选择所需要打开的数据文本文件，然后用只读和文本格式打开文件，逐行读取其内容，将每行字符串显示到界面上的 plainTextEdit 里，并且添加到一个临时的 QStringList 类型的变量 fFileContent 里。

然后调用自定义函数 iniModelFromStringList()，用 fFileContent 的内容初始化数据模型。下面是 iniModelFromStringList() 函数的代码：

```

void MainWindow::iniModelFromStringList(QStringList& aFileContent)
{ //从一个StringList 获取数据, 初始化数据模型
    int rowCnt=aFileContent.count(); //文本行数, 第1行是标题
    theModel->setRowCount(rowCnt-1);
//设置表头, 一个或多个空格、TAB 等分隔符隔开的字符串, 分解为一个StringList
    QString header=aFileContent.at(0); //第1行是表头
    QStringList headerList=
        header.split(QRegExp("\\s+"),QString::SkipEmptyParts);
    theModel->setHorizontalHeaderLabels(headerList); //设置表头文字
//设置表格数据
    QStandardItem *aItem;
    QStringList tmpList;
    int j;
    for (int i=1;i<rowCnt;i++)
    {
        QString aLineText=aFileContent.at(i);
        tmpList=aLineText.split(QRegExp("\\s+"),QString::SkipEmptyParts);
        for (j=0;j<FixedColumnCount-1;j++)
        { //不包含最后一列
            aItem=new QStandardItem(tmpList.at(j));
            theModel->setItem(i-1,j,aItem); //为模型的某个行列位置设置 Item
        }
        aItem=new QStandardItem(headerList.at(j)); //最后一列
        aItem->setCheckable(true); //设置为 Checkable
        if (tmpList.at(j)=="0")
            aItem->setCheckState(Qt::Unchecked);
        else
            aItem->setCheckState(Qt::Checked);
        theModel->setItem(i-1,j,aItem);
    }
}

```

传递来的参数 `aFileContent` 是文本文件所有行构成的 `StringList`, 文件的每一行是 `aFileContent` 的一行字符串, 第1行是表头文字, 数据从第2行开始。

程序首先获取字符串列表的行数, 然后设置数据模型的行数, 因为数据模型的列数在初始化时已经设置了。

然后获取字符串列表的第1行, 即表头文字, 用 `QString::split()` 函数分割成一个 `QStringList`, 设置为数据模型的表头标题。

`QString::split()` 函数根据某个特定的符号将字符串进行分割。例如, `header` 是数据列的标题, 每个标题之间通过一个或多个 `TAB` 键分隔, 其内容是:

```
测深 (m)      垂深 (m)      方位 (°)      总位移 (m)      固井质量      测井取样
```

那么通过上面的 `split()` 函数操作, 得到一个字符串列表 `headerList`, 其内容是:

```
测深 (m)
垂深 (m)
方位 (°)
总位移 (m)
固井质量
测井取样
```

也就是分解为一个 6 行的 `StringList`。然后使用此字符串列表作为数据模型, 设置表头标题的

函数 `setHorizontalHeaderLabels()` 的参数, 就可以为数据模型设置表头了。

同样, 在逐行获取字符串后, 也采用 `split()` 函数进行分解, 为每个数据创建一个 `QStandardItem` 类型的项数据 `aItem`, 并赋给数据模型作为某行某列的项数据。

`QStandardItemModel` 以二维表格的形式保存项数据, 每个项数据对应着 `QTableView` 的一个单元格。项数据不仅可以存储显示的文字, 还可以存储其他角色的数据。

数据文件的最后一列是一个逻辑型数据, 在 `tableView` 上显示时为其提供一个 `CheckBox` 组件, 此功能通过调用 `QStandardItem` 的 `setCheckable()` 函数实现。

### 3. 数据修改

当 `TableView` 设置为可编辑时, 双击一个单元格可以修改其内容, 对于使用 `CheckBox` 的列, 改变 `CheckBox` 的勾选状态, 就可以修改单元格关联项的选择状态。

在实例主窗口工具栏上有“添加行”“插入行”“删除行”按钮, 它们实现相应的编辑操作, 这些操作都是直接针对数据模型的, 数据模型被修改后, 会直接在 `TableView` 上显示出来。

#### • 添加行

“添加行”操作是在数据表的最后添加一行, 其实现代码如下:

```
void MainWindow::on_actAppend_triggered()
{ //在表格最后添加行
    QList<QStandardItem*> aItemList; //列表类
    QStandardItem *aItem;
    for(int i=0;i<FixedColumnCount-1;i++) //不包含最后1列
    {
        aItem=new QStandardItem("0"); //创建 Item
        aItemList<<aItem; //添加到列表
    }
    //获取最后一列的表头文字
    QString str=theModel->headerData(theModel->columnCount()-1, Qt::Horizontal,
Qt::DisplayRole).toString();
    aItem=new QStandardItem(str); //创建 "测井取样" Item
    aItem->setCheckable(true);
    aItemList<<aItem; //添加到列表

    theModel->insertRow(theModel->rowCount(),aItemList); //插入一行
    QModelIndex curIndex=theModel->index(theModel->rowCount()-1,0);
    theSelection->clearSelection();
    theSelection->setCurrentIndex(curIndex,QItemSelectionModel::Select);
}
```

使用 `QStandardItemModel::insertRow()` 函数插入一行, 其函数原型是:

```
void insertRow(int row, const QList<QStandardItem *> &items)
```

其中, `row` 是一个行号, 表示在此行号之前插入一行, 若 `row` 等于或大于总行数, 则在最后添加一行。`QList<QStandardItem *> &items` 是一个 `QStandardItem` 类型的列表类, 需要为插入的一行的每个项数据创建一个 `QStandardItem` 类型的项, 然后传递给 `insertRow()` 函数。

在这段程序中, 为前 5 列创建 `QStandardItem` 对象时, 都使用文字“0”, 最后一列使用表头的标题, 并设置为 `Checkable`。创建完每个项数据对象后, 使用 `insertRow()` 函数在最后添加一行。

- 插入行

“插入行”按钮的功能是在当前行的前面插入一行，实现代码与“添加行”类似。

- 删除行

“删除行”按钮的功能是删除当前行，首先从选择模型中获取当前单元格的模型索引，然后从模型索引中获取行号，调用 `removeRow(int row)` 删除指定的行。

```
void MainWindow::on_actDelete_triggered()
{ //删除行
    QModelIndex curIndex=theSelection->currentIndex(); //获取模型索引
    if (curIndex.row()==theModel->rowCount()-1) //最后一行
        theModel->removeRow(curIndex.row()); //删除最后一行
    else
    {
        theModel->removeRow(curIndex.row()); //删除一行，并重新设置当前选择行
        theSelection->setCurrentIndex(curIndex,QItemSelectionModel::Select);
    }
}
```

#### 4. 单元格格式设置

工具栏上有 3 个设置单元格文字对齐方式的按钮，还有一个设置字体粗体的按钮。当在 `TableView` 中选择多个单元格时，可以同时设置多个单元格的格式。例如，“居左”按钮的代码如下：

```
void MainWindow::on_actAlignLeft_triggered()
{ //设置文字居左对齐
    if (!theSelection->hasSelection())
        return;
    //获取选择的单元格的模型索引列表，可以是多选
    QModelIndexList selectedIndex=theSelection->selectedIndexes();
    for (int i=0;i<selectedIndex.count();i++)
    {
        QModelIndex aIndex=selectedIndex.at(i); //获取一个模型索引
        QStandardItem* aItem=theModel->itemFromIndex(aIndex);
        aItem->setTextAlignment(Qt::AlignLeft); //设置文字对齐方式
    }
}
```

`QItemSelectionModel::selectedIndexes()` 函数返回选择单元格的模型索引列表，然后通过此列表获取每个选择的单元格的模型索引，再通过模型索引获取其项数据，然后调用 `QStandardItem::setTextAlignment()` 设置一个项的对齐方式即可。

“居中”和“居右”按钮的代码与此类似。

“粗体”按钮设置单元格的字体是否为粗体，在选择单元格时，`actFontBold` 的 `check` 状态根据当前单元格的字体是否为粗体自动更新。`actFontBold` 的 `triggered(bool)` 的槽函数代码如下，与设置对齐方式的代码操作方式类似：

```
void MainWindow::on_actFontBold_triggered(bool checked)
{ //设置字体粗体
    if (!theSelection->hasSelection())
        return;
    QModelIndexList selectedIndex=theSelection->selectedIndexes();
    for (int i=0;i< selectedIndex.count();i++)
```



```

    {
        QModelIndex aIndex= selectedIndex.at(i); //获取一个模型索引
        QStandardItem* aItem=theModel->itemFromIndex(aIndex); //获取项数据
        QFont font=aItem->font();
        font.setBold(checked); //设置字体是否粗体
        aItem->setFont(font);
    }
}

```

### 5. 数据另存为文件

在视图组件上对数据的修改都会自动更新到数据模型里，单击工具栏上的“模型数据预览”按钮，可以将数据模型的数据内容显示到 PlainTextEdit 里。

数据模型里的数据是在内存中的，工具栏上的“另存文件”按钮可以将数据模型的数据另存为一个数据文本文件，同时也显示在 PlainTextEdit 里，其实现代码如下：

```

void MainWindow::on_actSave_triggered()
{ //保存为文件
    QString curPath=QCoreApplication::applicationDirPath();
    QString aFileName=QFileDialog::getSaveFileName(this,"选择一个文件", curPath,"井斜数
数据文件(*.txt);;所有文件(*.*)");
    if (aFileName.isEmpty())
        return;
    QFile aFile(aFileName);
    if (!(aFile.open(QIODevice::ReadWrite | QIODevice::Text | QIODevice::Truncate)))
        return; //以读写、覆盖原有内容方式打开文件

    QTextStream aStream(&aFile);
    QStandardItem *aItem;
    int i,j;
    QString str;
    ui->plainTextEdit->clear();
    //获取表头文字
    for (i=0;i<theModel->columnCount();i++)
    {
        aItem=theModel->horizontalHeaderItem(i); //获取表头的项数据
        str=str+aItem->text()+"\t\t"; //以 TAB 隔开
    }
    aStream<<str<<"\n"; //文件里需要加入换行符 \n
    ui->plainTextEdit->appendPlainText(str);
    //获取数据区文字
    for ( i=0;i<theModel->rowCount();i++)
    {
        str="";
        for( j=0;j<theModel->columnCount()-1;j++)
        {
            aItem=theModel->item(i,j);
            str=str+aItem->text()+QString::asprintf("\t\t");
        }
        aItem=theModel->item(i,j); //最后一列是逻辑型
        if (aItem->checkState()==Qt::Checked)
            str=str+"1";
        else
            str=str+"0";
        ui->plainTextEdit->appendPlainText(str);
        aStream<<str<<"\n";
    }
}

```

## 5.5 自定义代理

### 5.5.1 自定义代理的功能

在前一节的实例 samp5\_3 中，导入数据文件进行编辑时，QTableView 组件为每个单元格提供的是缺省的代理编辑组件，就是一个 QLineEdit 组件。在编辑框里可以输入任何数据，所以比较通用。但是有些情况下，希望根据数据的类型限定使用不同的编辑组件，例如在 samp5\_3 的实例的数据中，第 1 列“测深”是整数，使用 QSpinBox 作为编辑组件更合适；“垂深”“方位”“总位移”是浮点数，使用 QDoubleSpinBox 更合适；而“固井质量”使用一个 QComboBox，从一组列表文字中选择更合适。

要实现这些功能，就需要为 TableView 的某列或某个单元格设置自定义代理组件。本节在实例 samp5\_3 的基础上，为 TableView 增加自定义代理组件功能。设定自定义代理组件之后的 TableView 运行时，其编辑状态的效果如图 5-11 所示。

TableView

	测深(m)	垂深(m)	方位(°)	总位移(m)
1	252	252	241.27	0.51
2	270	275	241.27	0.72
3	300	300	235.82	0.9
4	325	325	239.23	1.09

图 5-11 设置自定义代理组件后的 TableView 编辑时的效果

### 5.5.2 自定义代理类的基本设计要求

Qt 中有关代理的几个类的层次结构如图 5-12 所示。

QAbstractItemDelegate 是所有代理类的抽象基类，QStyledItemDelegate 是视图组件使用的缺省的代理类，QItemDelegate 也是类似功能的类。QStyledItemDelegate 与 QItemDelegate 的差别在于：QStyledItemDelegate 可以使用当前的样式表设置来绘制组件，因此建议使用 QStyledItemDelegate 作为自定义代理组件的基类。

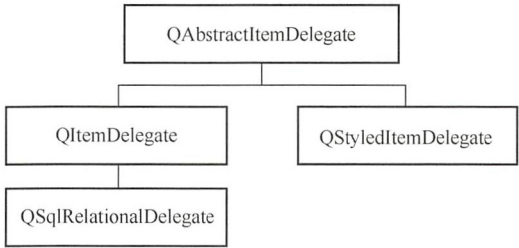


图 5-12 实现代理功能的类的层次结构

不管从 QStyledItemDelegate 还是 QItemDelegate 继承设计自定义代理组件，都必须实现如下的 4 个函数：

- createEditor()函数创建用于编辑模型数据的 widget 组件，如一个 QSpinBox 组件，或一个 QComboBox 组件；
- setEditorData()函数从数据模型获取数据，供 widget 组件进行编辑；
- setModelData()将 widget 上的数据更新到数据模型；
- updateEditorGeometry()用于给 widget 组件设置一个合适的大小。

### 5.5.3 基于 QSpinBox 的自定义代理类

#### 1. 自定义代理类的基本结构

下面设计一个基于 QSpinBox 类的自定义代理类，用于“测深”数据列的编辑。

在 Qt Creator 里单击“File”→“New File or Project”菜单项，在出现的“New File or Project”对话框里选择新建一个 C++class 文件，在出现的对话框里，输入自定义类的名称为 QWIntSpinDelegate，设置基类为 QStyledItemDelegate，单击下一步后结束向导，系统会自动生成头文件和源文件，并添加到项目里。

在头文件 qwintspindelegate.h 中包含对自定义类 QWIntSpinDelegate 的定义，在其中添加 4 个需要重定义的函数的定义，qwintspindelegate.h 的内容如下：

```
#include <QStyledItemDelegate>
class QWIntSpinDelegate : public QStyledItemDelegate
{
    Q_OBJECT
public:
    QWIntSpinDelegate(QObject *parent=0);
    //自定义代理组件必须继承以下 4 个函数
    QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem &option,
        const QModelIndex &index) const Q_DECL_OVERRIDE;
    void setEditorData(QWidget *editor,
        const QModelIndex &index) const Q_DECL_OVERRIDE;
    void setModelData(QWidget *editor, QAbstractItemModel *model,
        const QModelIndex &index) const Q_DECL_OVERRIDE;
    void updateEditorGeometry(QWidget *editor, const QStyleOptionViewItem
        &option, const QModelIndex &index) const Q_DECL_OVERRIDE;
};
```

自定义代理组件必须重新实现这 4 个函数，函数的原型都是固定的。

## 2. createEditor()函数的实现

createEditor()函数用于创建需要的编辑组件，QWIntSpinDelegate 类希望创建一个 QSpinBox 作为编辑组件，函数的实现如下：

```
QWidget *QWIntSpinDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem &option, const QModelIndex &index) const
{ //创建代理编辑组件
    QSpinBox *editor = new QSpinBox(parent);
    editor->setFrame(false); //设置为无边框
    editor->setMinimum(0);
    editor->setMaximum(10000);
    return editor; //返回此编辑器
}
```

这段代码创建了一个 QSpinBox 类型的编辑器 editor，parent 指向视图组件；然后对创建的 editor 做一些设置，将 editor 作为函数的返回值。

## 3. setEditorData()函数

setEditorData()函数用于从数据模型获取数值，设置为编辑器的显示值。当双击一个单元格进入编辑状态时，就会自动调用此函数，其实现代码如下：

```
void QWIntSpinDelegate::setEditorData(QWidget *editor, const QModelIndex &index) const
{ //从数据模型获取数据，显示到代理组件中
    int value = index.model()->data(index, Qt::EditRole).toInt();
    QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
    spinBox->setValue(value);
}
```

函数传递来的参数 `editor` 指向代理编辑组件，`index` 是关联的数据单元的模型索引。

通过强制类型转换将 `editor` 转换为 `QSpinBox` 类型组件 `spinBox`，然后将获取的数值设置为 `spinBox` 的值。

#### 4. `setModelData()` 函数

`setModelData()` 函数用于将代理编辑器上的值更新给数据模型，当用户在界面上完成编辑时会自动调用此函数，将界面上的数据更新到数据模型。其代码如下：

```
void QWIntSpinDelegate::setModelData(QWidget *editor, QAbstractItemModel *model, const
QModelIndex &index) const
{ //将代理组件的数据保存到数据模型中
    QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
    spinBox->interpretText();
    int value = spinBox->value();
    model->setData(index, value, Qt::EditRole);
}
```

程序先获取代理组件编辑器里的数值，然后利用传递来的数据模型 `model` 和模型索引参数 `index` 将编辑器的最新值更新到数据模型里。

#### 5. `updateEditorGeometry()` 函数

`updateEditorGeometry()` 函数用于为代理组件设置一个合适的大小，函数传递的参数 `option` 的 `rect` 变量定义了单元格适合显示代理组件的大小，直接设置为此值即可。代码如下。

```
void QWIntSpinDelegate::updateEditorGeometry(QWidget *editor, const QStyleOptionViewItem
&option, const QModelIndex &index) const
{ //设置组件大小
    editor->setGeometry(option.rect);
}
```

### 5.5.4 自定义代理类的使用

同样的，可以创建基于 `QDoubleSpinBox` 的自定义代理组件类 `QWFloatSpinDelegate`，用于编辑浮点数，还可以创建基于 `QComboBox` 的自定义组件类 `QWComboBoxDelegate`。在主窗口的类定义中定义 3 个代理类的实例变量（省略了其他定义内容）：

```
class MainWindow : public QMainWindow
{
private:
    QWIntSpinDelegate    intSpinDelegate; //整型数
    QWFloatSpinDelegate  floatSpinDelegate; //浮点数
    QWComboBoxDelegate  comboBoxDelegate; //列表选择
}
```

在 `MainWindow` 的构造函数中，为 `tableView` 的某些列设置自定义代理组件。增加了自定义代理组件的构造函数代码如下（去掉了初始化状态栏等一些不重要的内容）：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    theModel = new QStandardItemModel(2, FixedColumnCount, this);
    theSelection = new QItemSelectionModel(theModel); //选择模型
```



```

connect(theSelection, SIGNAL(currentChanged(QModelIndex, QModelIndex)),
        this, SLOT(on_currentChanged(QModelIndex, QModelIndex)));
ui->tableView->setModel(theModel); //设置数据模型
ui->tableView->setSelectionModel(theSelection); //设置选择模型
//为各列设置自定义代理组件
ui->tableView->setItemDelegateForColumn(0, &intSpinDelegate); //测深
ui->tableView->setItemDelegateForColumn(1, &floatSpinDelegate); //浮点数
ui->tableView->setItemDelegateForColumn(2, &floatSpinDelegate); //浮点数
ui->tableView->setItemDelegateForColumn(3, &floatSpinDelegate); //浮点数
ui->tableView->setItemDelegateForColumn(4, &comboBoxDelegate); //列表
}

```

为 TableView 的某一列设置自定义代理组件，使用 `setItemDelegateForColumn()` 函数；为某一行设置自定义代理组件，可使用 `setItemDelegateForRow()` 函数；若为整个 TableView 设置一个自定义代理组件，则调用 `setItemDelegate()` 函数。

如此增加了自定义代理功能后，在编辑“测深”列时，会在单元格的位置出现一个 `SpinBox` 组件，用于输入整数；在编辑第 2 至 4 列的浮点数时，会出现一个 `DoubleSpinBox` 组件，用于输入浮点数；而在编辑“固井质量”列时，会自动出现一个 `ComboBox` 组件，用于从下拉列表中选择一个字符串。

# 对话框与多窗体设计

在一个完整的应用程序设计中，不可避免地会涉及多个窗体、对话框的设计和调用，如何设计和调用这些对话框和窗体是搞清楚一个庞大的应用程序设计的基础。本章将介绍对话框和多窗体设计、调用方式、数据传递等问题，主要包括以下几点。

- Qt 提供的标准对话框的使用，如打开文件对话框、选择颜色对话框、字体对话框、消息提示和确认选择对话框等。
- 自定义对话框的设计和调用，如何获取返回值，在对话框中如何操作主窗体等。
- 在一个应用程序中如何设计多种窗体，基于 QDialog、QWidget 和 QMainWindow 创建的窗体的调用方式有哪些，它们之间有什么区别。
- 如何创建一个在分页组件中管理的多窗体应用，类似于现在流行的分页浏览器的界面效果，子窗体如何与主窗体实现交互。
- 如何创建 MDI（Multi-document interface）应用程序。
- 如何创建一个带有启动界面（Splash）和登录界面的窗体，如何保存和读取应用程序设置的参数。

## 6.1 标准对话框

### 6.1.1 概述

Qt 为应用程序设计提供了一些常用的标准对话框，如打开文件对话框、选择颜色对话框、信息提示和确认选择对话框、标准输入对话框等，用户无需再自己设计这些常用的对话框，这样可以减少程序设计工作量。在前面几章的实例中，或多或少地用到了其中的一些对话框。

Qt 预定义的各标准对话框的类，及其主要静态函数的功能见表 6-1（由于输入参数一般较多，省略了函数的输入参数，只列出了函数的返回值类型）。

表 6-1 Qt 预定义标准对话框

对话框	常用静态函数名称	函数功能
QFileDialog 文件对话框	QString getOpenFileName()	选择打开一个文件
	QStringList getOpenFileNames()	选择打开多个文件
	QString getSaveFileName()	选择保存一个文件
	QString getExistingDirectory()	选择一个已有的目录
	QUrl getOpenFileUrl()	选择打开一个文件，可选择远程网络文件

续表

对话框	常用静态函数名称	函数功能
QcolorDialog 颜色对话框	QColor getColor()	选择颜色
QFontDialog 字体对话框	QFont getFont()	选择字体
QInputDialog 输入对话框	QString getText()	输入单行文字
	int getInt()	输入整数
	double getDouble()	输入浮点数
	QString getItem()	从一个下拉列表框中选择输入
	QString getMultiLineText()	输入多行字符串
QMessageBox 消息框	StandardButton information()	信息提示对话框
	StandardButton question()	询问并获取是否确认的对话框
	StandardButton warning()	警告信息提示对话框
	StandardButton critical()	错误信息提示对话框
	void about()	设置自定义信息的关于对话框
	void aboutQt()	关于 Qt 的对话框

实例 samp6\_1 演示使用这些对话框，程序运行界面如图 6-1 所示。下方的文本框显示打开文件的文件名或一些提示信息，某些对话框的输入结果可应用于文本框的属性设置，如字体和颜色。



图 6-1 实例 samp6\_1 运行界面

### 6.1.2 QFileDialog 对话框

#### 1. 选择打开一个文件

若要打开一个文件，可调用静态函数 `QFileDialog::getOpenFileName()`，“打开一个文件”按钮的响应代码如下：

```
void Dialog::on_btnOpen_clicked()
{ //选择单个文件
    QString curPath=QDir::currentPath();//获取应用程序当前目录
    QString dlgTitle="选择一个文件";
    QString filter="文本文件(*.txt);;图片文件(*.jpg *.gif);;所有文件(*.*)";
    QString aFileName=QFileDialog::getOpenFileName(this,
        dlgTitle, curPath, filter);
    if (!aFileName.isEmpty())
        ui->plainTextEdit->appendPlainText(aFileName);
}
```

QFileDialog::getOpenFileName()函数需要传递3个字符串型参数, 分别如下。

- 对话框标题, 这里设置为“选择一个文件”。
- 初始化目录, 打开对话框时的初始目录, 这里用 QDir::currentPath() 获取应用程序当前目录。
- 文件过滤器, 设置选择不同后缀的文件, 可以设置多组文件, 如:

```
QString filter="文本文件(*.txt);;图片文件(*.jpg *.gif *.png);;所有文件(*.*)";
```

每组文件之间用两个分号隔开, 同一组内不同后缀之间用空格隔开。

QFileDialog::getOpenFileName()函数返回的是选择文件的带路径的完整文件名, 如果在对话框里取消选择, 则返回字符串为空。

## 2. 选择打开多个文件

若要选择打开多个文件, 可使用静态函数 QFileDialog::getOpenFileNames(), “打开多个文件”按钮的响应代码如下:

```
void Dialog::on_btnOpenMulti_clicked()
{ //选择多个文件
    QString curPath=QDir::currentPath();
    QString dlgTitle="选择多个文件";
    QString filter="文本文件(*.txt);;图片文件(*.jpg *.gif);;所有文件(*.*)";
    QStringList fileList=QFileDialog::getOpenFileNames(this,
        dlgTitle,curPath,filter);
    for (int i=0; i<fileList.count();i++)
        ui->plainTextEdit->appendPlainText(fileList.at(i));
}
```

getOpenFileNames()函数的传递参数与 getOpenFileName()一样, 只是返回值是一个字符串列表, 列表的每一行是选择的一个文件。

## 3. 选择已有目录

选择已有目录可调用静态函数 QFileDialog::getExistingDirectory(), 同样, 若需要传递对话框标题和初始路径, 还应传递一个选项, 一般用 QFileDialog::ShowDirsOnly, 表示对话框中只显示目录。

```
void Dialog::on_btnSelDir_clicked()
{ //选择文件夹
    QString curPath=QCoreApplication::applicationDirPath();
    QString dlgTitle="选择一个目录";
    QString selectedDir=QFileDialog::getExistingDirectory(this,
        dlgTitle,curPath, QFileDialog::ShowDirsOnly);
    if (!selectedDir.isEmpty())
        ui->plainTextEdit->appendPlainText(selectedDir);
}
```

静态函数 QCoreApplication::applicationDirPath() 返回应用程序可执行文件所在的目录, getExistingDirectory()函数的返回值是选择的目录名称字符串。

## 4. 选择保存文件名

选择一个保存文件, 使用静态函数 QFileDialog::getSaveFileName(), 传递的参数与 getOpenFileName()函数相同。只是在调用 getSaveFileName()函数时, 若选择的是一个已经存在的文件, 会提



示是否覆盖原有的文件。如果提示覆盖，会返回为选择的文件，但是并不会对文件进行实质操作，对文件的删除操作需要在选择文件之后自己编码实现。如下面的代码，即使选择覆盖文件，由于代码里没有实质地覆盖原来的文件，也不会对选择的文件造成任何影响。

```
void Dialog::on_btnSave_clicked()
{ //保存文件
    QString curPath=QCoreApplication::applicationDirPath();
    QString dlgTitle="保存文件";
    QString filter="h 文件 (*.h);;C++文件 (.cpp);;所有文件 (*.*)";
    QString aFileName=QFileDialog::getSaveFileName(this,
        dlgTitle,curPath,filter);
    if (!aFileName.isEmpty())
        ui->plainTextEdit->appendPlainText(aFileName);
}
```

### 6.1.3 QColorDialog 对话框

QColorDialog 是选择颜色对话框，选择颜色使用静态函数 QColorDialog::getColor()。下面是“选择颜色”按钮的代码，它为文本框的字体选择颜色。

```
void Dialog::on_btnColor_clicked()
{ //选择颜色
    QPalette pal=ui->plainTextEdit->palette(); //获取现有 palette
    QColor iniColor=pal.color(QPalette::Text); //现有的文字颜色
    QColor color=QColorDialog::getColor(iniColor,this,"选择颜色");
    if (color.isValid())
    {
        pal.setColor(QPalette::Text,color);
        ui->plainTextEdit->setPalette(pal);
    }
}
```

getColor()函数需要传递一个初始的颜色，这里是将 palette 提取的文本颜色作为初始颜色。getColor()函数返回一个颜色变量，若在颜色对话框里取消选择，则返回的颜色值无效，通过 QColor::isValid()函数来判断返回是否有效。

### 6.1.4 QFontDialog 对话框

QFontDialog 是选择字体对话框，选择字体使用静态函数 QFontDialog::getFont()。下面是“选择字体”按钮的代码，它为文本框选择字体，字体设置的内容包括字体名称、大小、粗体、斜体等。

```
void Dialog::on_btnFont_clicked()
{ //选择字体
    QFont iniFont=ui->plainTextEdit->font();
    bool ok=false;
    QFont font=QFontDialog::getFont(&ok,iniFont);
    if (ok)
        ui->plainTextEdit->setFont(font);
}
```

getFont()返回一个字体变量，但是 QFont 没有类似于 isValid()的函数来判断有效性，所以在调

用 `getFont()` 函数时以引用方式传递一个逻辑变量 `ok`，调用后通过判断 `ok` 是否为 `true` 来判断字体选择是否有效。

### 6.1.5 QLineEdit 标准输入对话框

`QInputDialog` 有单行字符串输入、整数输入、浮点数输入、列表框选择输入和多行文本等多种输入方式，图 6-2 是其中 4 种界面效果。



图 6-2 QLineEdit 4 种输入对话框

#### 1. 输入文字

`QInputDialog::getText()` 函数显示一个对话框用于输入字符串，传递的参数包括对话框标题、提示标签文字、缺省输入、编辑框响应模式等。

其中编辑框响应模式是枚举类型 `QLineEdit::EchoMode`，它控制编辑框上文字的显示方式，正常情况下选择 `QLineEdit::Normal`；如果是输入密码，选择 `QLineEdit::Password`。代码如下：

```
void Dialog::on_btnInputString_clicked()
{ //输入字符串
    QString dlgTitle="输入文字对话框";
    QString txtLabel="请输入文件名";
    QString defaultInput="新建文件.txt";
    QLineEdit::EchoMode echoMode=QLineEdit::Normal;
    // QLineEdit::EchoMode echoMode=QLineEdit::Password; //密码输入
    bool ok=false;
    QString text = QInputDialog::getText(this, dlgTitle,txtLabel,
        echoMode,defaultInput, &ok);
    if (ok && !text.isEmpty())
        ui->plainTextEdit->appendPlainText(text);
}
```

#### 2. 输入整数

使用 `QInputDialog::getInt()` 函数输入一个整数，下面的代码为文本选择字体大小。

```
void Dialog::on_btnInputInt_clicked()
{//输入整数
    QString dlgTitle="输入整数对话框";
    QString txtLabel="设置字体大小";
    int defaultValue=ui->plainTextEdit->font().pointSize();
    int minValue=6, maxValue=50,stepValue=1;
```

```

bool ok=false;
int inputValue = QInputDialog::getInt(this, dlgTitle,txtLabel,
                                     defaultValue, minValue,maxValue,stepValue,&ok);

if (ok)
{
    QFont    font=ui->plainTextEdit->font();
    font.setPointSize(inputValue);
    ui->plainTextEdit->setFont(font);
}
}

```

输入整数对话框使用一个 SpinBox 组件输入整数, getInt()需要传递的参数包括数值大小范围、步长、初始值, 确认选择输入后, 将输入的整数值作为文本框字体的大小。

### 3. 输入浮点数

使用 QInputDialog::getDouble()函数输入一个浮点数, 输入对话框使用一个 QDoubleSpinBox 作为输入组件, getDouble()的输入参数需要输入范围、初始值、小数点位数等。代码如下:

```

void Dialog::on_btnInputFloat_clicked()
{ //输入浮点数
    QString dlgTitle="输入浮点数对话框";
    QString txtLabel="输入一个浮点数";
    float defaultValue=3.13;
    float minValue=0, maxValue=10000;
    int decimals=2;//小数点位数
    bool ok=false;
    float inputValue = QInputDialog::getDouble(this, dlgTitle,txtLabel,
                                              defaultValue, minValue,maxValue,decimals,&ok);

    if (ok)
    {
        QString str=QString::asprintf("输入了一个浮点数:%.2f",inputValue);
        ui->plainTextEdit->appendPlainText(str);
    }
}

```

### 4. 下拉列表选择输入

使用 QInputDialog::getItem()可以从一个 ComboBox 组件的下拉列表中选择输入。代码如下:

```

void Dialog::on_btnInputItem_clicked()
{ //条目选择输入
    QStringList items;
    items <<"优秀"<<"良好"<<"合格"<<"不合格";
    QString dlgTitle="条目选择对话框";
    QString txtLabel="请选择级别";
    int    curIndex=0; //初始选择项
    bool    editable=true; //ComboBox 是否可编辑
    bool    ok=false;
    QString text = QInputDialog::getItem(this, dlgTitle, txtLabel,
                                         items, curIndex, editable, &ok);

    if (ok && !text.isEmpty())
        ui->plainTextEdit->appendPlainText(text);
}

```

getItem()函数需要一个 QStringList 变量为其 ComboBox 组件做条目初始化, curIndex 指明初始选择项, editable 表示对话框里的 ComboBox 是否可编辑, 若不能编辑, 则只能在下拉列表中选择。

## 6.1.6 QMessageBox 消息对话框

### 1. 简单信息提示

消息对话框 `QMessageBox` 用于显示提示、警告、错误等信息，或进行确认选择，由几个静态函数实现这些功能（详见表 6-1）。其中 `warning()`、`information()`、`critical()` 和 `about()` 这几个函数的输入参数和使用方法相同，只是信息提示的图标有区别。例如，`warning()` 的函数原型是：

```
StandardButton QMessageBox::warning(QWidget *parent, const QString &title, const
QString &text, StandardButtons buttons = Ok, StandardButton defaultButton = NoButton)
```

其中，`parent` 是对话框的父窗口，指定父窗口之后，打开对话框时，对话框将自动显示在父窗口的上方中间位置；`title` 是对话框标题字符串；`text` 是对话框需要显示的信息字符串；`buttons` 是对话框提供的按钮，缺省只有一个 OK 按钮；`defaultButton` 是缺省选择的按钮，缺省表示没有选择。

`warning()` 函数的返回结果是 `StandardButton` 类型。对话框上显示的按钮和缺省选中按钮也是 `StandardButton` 类型。

`StandardButton` 是各种按钮的定义，如 OK、Yes、No、Cancel 等，其枚举取值是 `QMessageBox::Ok`、`QMessageBox::Cancel`、`QMessageBox::Close` 等，详见 Qt 帮助文档中的 `StandardButton` 类型的说明。

对于 `warning()`、`information()`、`critical()` 和 `about()` 这几种对话框，它们一般只有一个 OK 按钮，且无须关心对话框的返回值。所以，使用缺省的按钮设置即可。例如，下面是程序中调用 `QMessageBox` 信息显示的代码，显示的几个对话框如图 6-3 所示。



图 6-3 QMessageBox 的几种消息提示对话框

```
void Dialog::on_btnMsgInformation_clicked()
{
    //information
    QString dlgTitle="information 消息框";
    QString strInfo="文件已经打开，字体大小已设置";
    QMessageBox::information(this, dlgTitle, strInfo,
        QMessageBox::Ok, QMessageBox::NoButton);
}

void Dialog::on_btnMsgWarning_clicked()
{
    // warning
    QString dlgTitle="warning 消息框";
    QString strInfo="文件内容已经被修改";
    QMessageBox::warning(this, dlgTitle, strInfo);
}

void Dialog::on_btnMsgCritical_clicked()
```



```

{ // critical
    QString dlgTitle="critical 消息框";
    QString strInfo="有不明程序访问网络";
    QMessageBox::critical(this, dlgTitle, strInfo);
}
void Dialog::on_btnMsgAbout_clicked()
{ // about
    QString dlgTitle="about 消息框";
    QString strInfo="我开发的数据查看软件 V1.0 \n 保留所有版权";
    QMessageBox::about(this, dlgTitle, strInfo);
}

```

## 2. 确认选择对话框

QMessageBox::question()函数用于打开一个选择对话框，提示信息，并提供 Yes、No、OK、Cancel 等按钮，用户单击某个按钮返回选择，如常见的文件保存确认对话框如图 6-4 所示。

静态函数 QMessageBox::question()的原型如下：

```

StandardButton QMessageBox::question(QWidget *parent, const QString &title, const
QString &text, StandardButtons buttons = StandardButtons( Yes | No ), StandardButton
defaultButton = NoButton)

```

question()对话框的关键是在其中可以选择显示多个按钮，例如同时显示 Yes、No、OK 或 Cancel。其返回结果也是一个 StandardButton 类型变量，表示哪个按钮被单击了。下面是产生如图 6-4 所示对话框的代码，并根据对话框选择结果进行了判断和显示。

```

void Dialog::on_btnMsgQuestion_clicked()
{
    QString dlgTitle="Question 消息框";
    QString strInfo="文件已被修改，是否保存修改? ";
    QMessageBox::StandardButton defaultBtn=QMessageBox::NoButton;
    QMessageBox::StandardButton result;//返回选择的按钮
    result=QMessageBox::question(this, dlgTitle, strInfo,
                                QMessageBox::Yes|QMessageBox::No |QMessageBox::Cancel,
                                defaultBtn);
    if (result==QMessageBox::Yes)
        ui->plainTextEdit->appendPlainText("Question 消息框: Yes 被选择");
    else if (result==QMessageBox::No)
        ui->plainTextEdit->appendPlainText("Question 消息框: No 被选择");
    else if (result==QMessageBox::Cancel)
        ui->plainTextEdit->appendPlainText("Question 消息框: Cancel 被选择");
    else
        ui->plainTextEdit->appendPlainText("Question 消息框: 无选择");
}

```

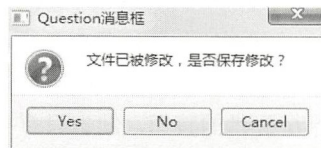


图 6-4 QMessageBox::question()生成的对话框

## 6.2 自定义对话框及其调用

### 6.2.1 对话框的不同调用方式

在一个应用程序设计中，为了实现一些特定的功能，必须设计自定义对话框。自定义对话框

的设计一般从 QDialog 继承，并且可以采用 UI 设计器可视化地设计对话框。对话框的调用一般包括创建对话框、传递数据给对话框、显示对话框获取输入、判断对话框单击按钮的返回类型、获取对话框输入数据等过程。

本节将通过实例 samp6\_2 来详细介绍这些原理。图 6-5 是实例 samp6\_2 的主窗口，及其设置表格行列数的对话框。

主窗口采用 QTableView 和 QStandardItemModel、QItemSelectionModel 构成一个通用的数据表格编辑器，设计了 3 个对话框，分别具有不同的功能，并且展示对话框不同调用方式的特点。

● 设置表格行列数对话框 QWDialogSize

该对话框每次动态创建，以模态方式显示（必须关闭此对话框才可以返回主窗口操作），对话框关闭后获取返回值，用于设置主窗口的表格行数和列数，并且删除对话框对象，释放内存。

这种对话框创建和调用方式适用于比较简单，不需要从主窗口传递大量数据做初始化的对话框，调用后删除对话框对象可以节约内存。

● 设置表头标题对话框 QWDialogHeaders

图 6-6 是设置表格表头标题的对话框，该对话框在父窗口（本例中就是主窗口）存续期间只创建一次，创建时传递表格表头字符串列表给对话框，在对话框里编辑表头标题后，主窗口获取编辑之后的表头标题。对话框以模态方式显示，关闭后只是隐藏，并不删除对象，下次再调用时只是打开已创建的对话框对象。

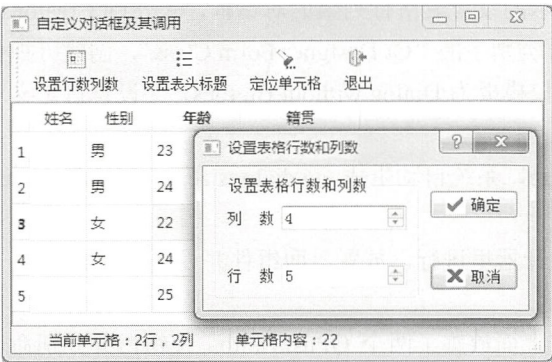


图 6-5 实例 samp6\_2 主窗口及其设置表格行列数的对话框



图 6-6 设置表格表头标题对话框

这种创建和调用方式适用于比较复杂的对话框，需要从父窗口传递大量数据做对话框初始化。下次调用时不需要重复初始化，能提高对话框调用速度，但是会一直占用内存，直到父窗口删除时，对话框才从内存中删除。

● 单元格定位与文字设置对话框 QWDialogLocate

图 6-7 是单元格定位和文字设置对话框，该对话框以非模态方式调用，显示对话框时还可以对主窗口进行操作，对话框只是浮动在窗口上方。在对话框里可以定位主窗口表格的某个单元格并设置其文字内容，在主窗口上的表格中单击鼠标时，单元格的行号、列号也会更新在对话框中。对话框关闭后将自动删除，释放内存。

这种对话框适用于主窗口与对话框需要交互操作的情况，例如用于查找和替换操作的对话框。

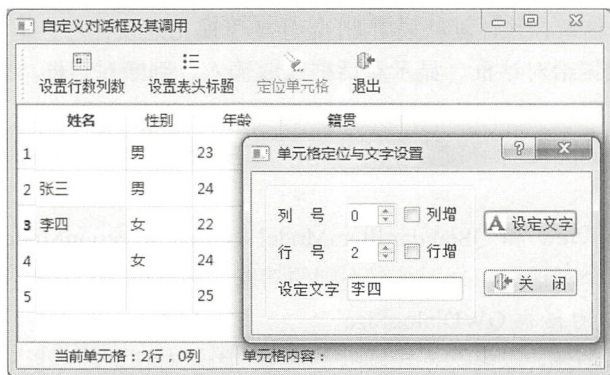


图 6-7 浮动于主窗口上方的对话框，可交互操作

## 6.2.2 对话框 QWDialogSize 的创建和使用

### 1. 创建对话框 QWDialogSize

实例主窗口从 `QMainWindow` 继承，主窗口用一个 `QTableView` 组件作为界面中心组件，设计几个 `Action` 用于创建主工具栏按钮。主窗口采用 `QStandardItemModel` 作为数据模型，`QItemSelectionModel` 作为选择模型，界面和主窗口 `Model/View` 结构的设计可参考 5.4 节的实例，本节不再详述。

在项目主窗口建立后，要创建如图 6-5 所示的设置表格行列数的对话框，单击 Qt Creator 的菜单项 “File” → “New File or Project”，选择 Qt 类别下的 “Qt Designer Form Class”，创建可视化设计的对话框类。在随后出现的向导里，选择窗口模板为 `Dialog without Buttons`，并设置自定义对话框的类名。

设置创建的对话框类名称为 `QWDialogSize`，系统自动生成 `qwdialogsize.h`、`qwdialogsize.cpp` 和 `qwdialogsize.ui` 3 个文件。

`QWDialogSize` 对话框的界面设计在 UI 设计器里进行，放置界面组件并设置好布局。

### 2. 对话框的调用和返回值

设计 `QWDialogSize` 对话框的界面时，在上面放置了两个 `QPushButton` 按钮，并分别命名为 `btnOK` 和 `btnCancel`，分别是“确定”和“取消”按钮，用于获取对话框运行时用户的选择。那么，如何获得用户操作的返回值呢？

在信号与槽编辑器里，将 `btnOK` 的 `clicked()` 信号与对话框的 `accept()` 槽关联，将 `btnCancel` 的 `clicked()` 信号与对话框的 `reject()` 槽关联即可，如图 6-8 所示。

单击“确定”按钮会执行 `accept()` 槽（或在代码里调用 `accept()` 槽函数也是一样的），这会关闭对话框（默认情况下，对话框只是被隐藏，并不被删除），并返回 `QDialog::Accepted` 作为 `exec()` 函数的返回值。

单击“取消”按钮会执行 `reject()` 槽函数，也会关闭对话框，并返回 `QDialog::Rejected` 作为 `exec()` 函数的返回值。

Sender	Signal	Receiver	Slot
btnOK	clicked()	QWDialogSize	accept()
btnCancel	clicked()	QWDialogSize	reject()

图 6-8 对话框设计时“确定”和“取消”按钮的信号与槽关联



完成后的 QWDialogSize 的类完整定义如下：

```
class QWDialogSize : public QDialog
{
    Q_OBJECT
public:
    explicit QWDialogSize(QWidget *parent = 0);
    ~QWDialogSize();
    int    rowCount(); //返回对话框输入的行数
    int    columnCount(); //返回对话框输入的列数
    void    setRowColumn(int row, int column); //初始对话框上两个 SpinBox 的值
private:
    Ui::QWDialogSize *ui;
};
```

在 QWDialogSize 的类定义中定义 3 个 public 函数，用于与对话框调用者的数据交互。因为窗体上的组件都是私有成员，外界不能直接访问界面组件，只能通过接口函数访问。

下面是类的接口函数实现代码。在析构函数中弹出一个消息提示对话框，以便观察对话框是何时被删除的。

```
QWDialogSize::~QWDialogSize()
{
    QMessageBox::information(this, "提示", "设置表格行列数对话框被删除");
    delete ui;
}
int QWDialogSize::rowCount()
{ //用于主窗口调用获得行数的输入值
    return ui->spinBoxRow->value();
}
int QWDialogSize::columnCount()
{ //用于主窗口调用获得列数的输入值
    return ui->spinBoxColumn->value();
}
void QWDialogSize::setRowColumn(int row, int column)
{ //初始化数据显示
    ui->spinBoxRow->setValue(row);
    ui->spinBoxColumn->setValue(column);
}
```

下面是主窗口中的“设置行数列数”工具栏按钮的响应代码，用于创建、显示对话框，并读取对话框上设置的行数、列数。

```
void MainWindow::on_actTab_SetSize_triggered()
{ //模态对话框，动态创建，用过删除
    QWDialogSize *dlgTableSize=new QWDialogSize(this);
    Qt::WindowFlags flags=dlgTableSize->windowFlags();
    dlgTableSize->setWindowFlags(flags | Qt::MSWindowsFixedSizeDialogHint);
    dlgTableSize->setRowColumn(theModel->rowCount(),
                              theModel->columnCount());
    int ret=dlgTableSize->exec(); // 以模态方式显示对话框，
    if (ret==QDialog::Accepted)
    { //OK 按钮被按下，获取对话框上的输入，设置行数和列数
        int cols=dlgTableSize->columnCount();
        theModel->setColumnCount(cols);
        int rows=dlgTableSize->rowCount();
        theModel->setRowCount(rows);
    }
```



```

    }
    delete dlgTableSize;
}

```

从代码中可以看到，每次单击此工具栏按钮时，对话框都被重新创建。创建后用 `QDialog` 的 `setWindowFlags()` 函数将对话框设置为固定大小，然后调用对话框的自定义函数 `setRowColumn()`，将主窗口数据模型 `theModel` 的现有的行数和列数显示到对话框上的两个 `SpinBox` 组件里。

调用对话框的 `exec()` 函数，以模态显示的方式显示对话框。模态显示方式下，用户只能在对话框上操作，不能操作主窗口，主程序也在此处等待 `exec()` 函数的返回结果。

当用户单击“确定”按钮关闭对话框后，`exec()` 返回结果为 `QDialog::Accepted`，主程序获得此返回结果后，通过对话框的自定义函数 `columnCount()` 和 `rowCount()` 获得对话框上新输入的列数和行数，然后设置为数据模型的列数和行数。

最后使用 `delete` 删除创建的对话框对象，释放内存。所以，关闭对话框时，会出现 `QWDialogSize` 析构函数里的消息提示对话框。

---

**注意** 在对话框上单击按钮或关闭对话框时，对话框只是隐藏（缺省的），而并没有从内存中删除。如果对话框一关闭就自动删除，则在后面调用对话框的自定义函数获得输入的行数和列数时会出现严重错误。

---

## 6.2.3 对话框 `QWDialogHeaders` 的创建和使用

### 1. 对话框的生存期

对话框的生存期是指它从创建到删除的存续区间。前面介绍的设置表格行数和列数的对话框的生存期只在调用它的按钮的槽函数里，因为对话框是动态创建的，调用结束后就会被删除。

而对于图 6-6 所示的设置表头标题对话框，我们希望在主窗口里首次调用时创建它，对话框关闭时并不删除，只是隐藏，下次调用时再次显示此对话框。只有在主窗口释放时该对话框才释放，所以这个对话框的生存期在主窗口存续期间。

### 2. `QWDialogHeaders` 的定义和实现

设置表头标题的对话框类是 `QWDialogHeaders`，它也是从 `QDialog` 继承的可视对话框类。其界面显示使用 `QListView` 组件，用 `QStringListModel` 变量管理字符串列表数据，构成 Model/View 结构。对话框上同样有“确定”和“取消”两个按钮，设置与对话框的 `accept()` 和 `reject()` 槽关联。

`QWDialogHeaders` 类的定义如下：

```

class QWDialogHeaders : public QDialog
{
    Q_OBJECT
private:
    QStringListModel *model;
public:
    explicit QWDialogHeaders(QWidget *parent = 0);
    ~QWDialogHeaders();
    void setHeaderList(QStringList& headers);
    QStringList headerList();
private:
    Ui::QWDialogHeaders *ui;
}

```

```
};
```

QWDialogSize 类接口函数实现的代码如下:

```
QWDialogHeaders::QWDialogHeaders(QWidget *parent) : QDialog(parent),
    ui(new Ui::QWDialogHeaders)
{
    ui->setupUi(this);
    model= new QStringListModel;
    ui->listView->setModel(model);
}
QWDialogHeaders::~QWDialogHeaders()
{
    QMessageBox::information(this, "提示", "设置表头标题对话框被删除");
    delete ui;
}
void QWDialogHeaders::setHeaderList(QStringList &headers)
{ //初始化数据模型的字符串列表
    model->setStringList(headers);
}
QStringList QWDialogHeaders::headerList()
{ //返回字符串列表
    return model->stringList();
}
```

### 3. QWDialogHeaders 对话框的使用

因为要在主窗口中重复调用此对话框,所以在 MainWindow 的 private 部分定义一个 QWDialogHeaders 类型的指针变量,并且将此指针初始化设置为 NULL,用于判断对话框是否已经被创建。在 MainWindow 中的定义如下:

```
private:
    QWDialogHeaders *dlgSetHeaders=NULL;
```

下面是主窗口工具栏上的“设置表头标题”按钮的响应代码。

```
void MainWindow::on_actTab_SetHeader_triggered()
{ //一次创建,多次调用,对话框关闭时只是隐藏
    if (dlgSetHeaders==NULL)
        dlgSetHeaders = new QWDialogHeaders(this);
    if (dlgSetHeaders->headerList().count()!=theModel->columnCount())
    { //如果表头列数变化,重新初始化
        QStringList strList;
        for (int i=0;i<theModel->columnCount();i++) //获取现有的表头标题
            strList.append(theModel->headerData(i,Qt::Horizontal,
                Qt::DisplayRole).toString());
        dlgSetHeaders->setHeaderList(strList); //对话框初始化显示
    }

    int ret=dlgSetHeaders->exec(); //以模态方式显示对话框
    if (ret==QDialog::Accepted) //OK 键被按下
    {
        QStringList strList=dlgSetHeaders->headerList();
        theModel->setHorizontalHeaderLabels(strList); //设置模型的表头标题
    }
}
```

在这段代码中，首先判断主窗口的成员变量 `dlgSetHeaders` 是否为 `NULL`，如果为 `NULL`（初始化为 `NULL`），说明对话框还没有被创建，就创建对话框。

初始化的工作是获取主窗口数据模型现有的表头标题，然后调用对话框的自定义函数 `setHeaderList()`，设置其为对话框的数据源。

使用 `exec()` 函数模态显示对话框，然后在“确定”按钮被单击时获取对话框上输入的字符串列表，设置为主窗口数据模型的表头标题。

---

**注意** 这里在结束对话框操作后，并没有使用 `delete` 操作删除对话框对象，这样对话框就只是隐藏，它还在内存中。关闭对话框时不会出现析构函数里的消息提示对话框。

---

对话框创建时，传递主窗口的指针作为对话框的父对象，即：

```
dlgSetHeaders = new QWDialogHeaders(this);
```

所以，主窗口释放时才会自动删除此对话框对象，也就是程序退出时才删除此对话框，才会出现 `QWDialogHeaders` 析构函数里的消息提示对话框。

## 6.2.4 对话框 `QWDialogLocate` 的创建与使用

### 1. 非模态对话框

前面设计的两个对话框是以模态（Modal）方式显示的，即用 `QDialog::exec()` 函数显示。模态显示的对话框不允许鼠标再去单击其他窗口，直到对话框退出。

若使用 `QDialog::show()`，则能以非模态（Modeless）方式显示对话框。非模态显示的对话框在显示后继续运行主程序，还可以在主窗口上操作，主窗口和非模态对话框之间可以交互控制，典型的例子是文字编辑软件里的“查找/替换”对话框。

图 6-7 中的单元格定位与文字设置对话框以非模态方式显示，对话框类是 `QWDialogLocate`，它有如下的一些功能。

- 主窗口每次调用此对话框时，就会创建此对话框对象，并以 `StayOnTop` 的方式显示，对话框关闭时自动删除。
- 在对话框中可以定位主窗口上 `TableView` 组件的单元格，并设置单元格的文字。
- 在主窗口的 `TableView` 组件中单击鼠标时，如果对话框已创建，则自动更新对话框上单元格的行号和列号 `SpinBox` 组件的值。
- 主窗口上的 `actTab_Locate` 用于调用对话框，调用时 `actTab_Locate` 设置为禁用，当对话框关闭时自动使能 `actTab_Locate`。这样避免对话框显示时，在主窗口上再次单击“定位单元格”按钮，而在对话框关闭和释放后，按钮又恢复为可用。

对话框 `QWDialogLocate` 的类定义代码如下，各接口函数的意义和实现在后面介绍。

```
class QWDialogLocate : public QDialog
{
    Q_OBJECT
private:
    void closeEvent(QCloseEvent *event);
```

```
public:
    explicit QWDialogLocate(QWidget *parent = 0);
    ~QWDialogLocate();
    void setSpinRange(int rowCount, int colCount); //设置最大值
    void setSpinValue(int rowNo, int colNo); //设置初始值
private slots:
    void on_btnSetText_clicked();
private:
    Ui::QWDialogLocate *ui;
};
```

## 2. 对话框的创建与调用

对话框 QWDialogLocate 是从 QDialog 继承而来的可视化设计的对话框类, 其界面设计不再详述。为了在主窗口中也能操作对话框, 需要保留对话框实例对象名, 所以在 MainWindow 定义对话框 QWDialogLocate 的一个指针 dlgLocate, 并初始化为 NULL。

```
private:
    QWDialogLocate *dlgLocate=NULL;
```

主窗口上的 actTab\_Locate 用于调用此对话框, 其 triggered() 信号槽函数代码如下:

```
void MainWindow::on_actTab_Locate_triggered()
{ //创建 StayOnTop 的对话框, 对话框关闭时自动删除
    ui->actTab_Locate->setEnabled(false);
    dlgLocate = new QWDialogLocate(this);
    dlgLocate->setAttribute(Qt::WA_DeleteOnClose); //对话框关闭时自动删除
    Qt::WindowFlags flags=dlgLocate->windowFlags(); //获取已有 flags
    dlgLocate->setWindowFlags(flags | Qt::WindowStaysOnTopHint); //StayOnTop

    dlgLocate->setSpinRange(theModel->rowCount(), theModel->columnCount());
    QModelIndex curIndex=theSelection->currentIndex();
    if (curIndex.isValid())
        dlgLocate->setSpinValue(curIndex.row(), curIndex.column());
    dlgLocate->show(); //非模态显示对话框
}
```

在这段代码中, 使用 QWidget::setAttribute() 函数将对话框设置为关闭时自动删除。

```
dlgLocate->setAttribute(Qt::WA_DeleteOnClose);
```

setAttribute() 用于对窗体的一些属性进行设置, 当设置为 Qt::WA\_DeleteOnClose 时, 窗口关闭时会自动删除, 以释放内存。这与前面两个对话框是不同的, 前面两个对话框在关闭时缺省是隐藏自己, 除非显式地使用 delete 进行删除。

程序还调用 QWidget::setWindowFlags() 将对话框设置为 StayOnTop 显示。

```
dlgLocate->setWindowFlags(flags | Qt::WindowStaysOnTopHint);
```

对话框窗口效果设置后, 再设置其初始数据, 然后调用 show() 显示对话框。显示对话框后, 主程序继续运行, 不会等待对话框的返回结果。鼠标可以操作主窗口上的界面, 但是因为 actTab\_Locate 被禁用了, 不能再重复单击“定位单元格”按钮。

## 3. 对话框中操作主窗口

在对话框上单击“设定文字”按钮, 会在主窗口中定位到指定的单元格, 并设定为输入的文字, 按钮的代码如下:



```

void QWDialogLocate::on_btnSetText_clicked()
{
    //定位到单元格, 并设置字符串
    int row=ui->spinBoxRow->value(); //行号
    int col=ui->spinBoxColumn->value(); //列号
    MainWindow *parWind = (MainWindow*)parentWidget(); //获取主窗口
    parWind->setACellText(row,col,ui->edtCaption->text()); //调用主窗口函数
    if (ui->chkBoxRow->isChecked()) //行增
        ui->spinBoxRow->setValue(1+ui->spinBoxRow->value());
    if (ui->chkBoxColumn->isChecked()) //列增
        ui->spinBoxColumn->setValue(1+ui->spinBoxColumn->value());
}

```

想要在对话框中操作主窗口, 就需要获取主窗口对象, 调用主窗口的函数并传递参数。在上面的代码中, 通过下面一行语句获得主窗口对象:

```
MainWindow *parWind = (MainWindow*)parentWidget();
```

`parentWidget()`是 `QWidget` 类的一个函数, 指向父窗口。在创建此对话框时, 将主窗口的指针传递给对话框的构造函数, 即:

```
dlgLocate = new QWDialogLocate(this);
```

所以, 对话框的 `parentWidget` 指向主窗口。

然后调用主窗口的一个自定义的 `public` 函数 `setACellText()`, 传递行号、列号和字符串, 由主窗口更新指定单元格的文字。下面是主窗口的 `setACellText()`函数的代码。

```

void MainWindow::setACellText(int row, int column, QString text)
{
    //定位到单元格, 并设置字符串
    QModelIndex index=theModel->index(row,column); //获取模型索引
    theSelection->clearSelection();
    theSelection->setCurrentIndex(index,QItemSelectionModel::Select);
    theModel->setData(index,text,Qt::DisplayRole); //设置单元格字符串
}

```

这样就实现了在对话框里对主窗口进行的操作, 主要是获取主窗口对象, 然后调用相应的函数。

#### 4. 主窗口中操作对话框

在主窗口上用鼠标单击 `TableView` 组件的某个单元格时, 如果单元格定位对话框 `dlgLocate` 已经存在, 就将单元格的行号、列号更新到对话框上, 实现代码如下:

```

void MainWindow::on_tableView_clicked(const QModelIndex &index)
{
    //单击单元格时, 将单元格的行号、列号设置到对话框上
    if (dlgLocate!=NULL)
        dlgLocate->setSpinValue(index.row(),index.column());
}

```

因为主窗口中定义了对话框的指针, 只要它不为 `NULL`, 就说明对话框存在, 调用对话框的一个自定义函数 `setSpinValue()`, 刷新对话框显示界面。`QWDialogLocate` 的 `setSpinValue()`函数实现如下:

```

void QWDialogLocate::setSpinValue(int rowNo, int colNo)
{
    //设置 SpinBox 组件的数值
    ui->spinBoxRow->setValue(rowNo);
    ui->spinBoxColumn->setValue(colNo);
}

```

## 5. 窗口的 CloseEvent 事件

对话框和主窗口之间互相操作的关键是要有对方对象的指针，然后才能传递参数并调用对方的函数。在对话框关闭时，还需要做一些处理：将主窗口的 `actTab_Locate` 重新设置为使能，将主窗口的指向对话框的指针 `dlgLocate` 重新设置为 `NULL`。

由于对话框 `dlgLocate` 是以非模态方式运行的，程序无法等待对话框结束后作出响应，但是可以利用窗口的 `CloseEvent` 事件。

事件（event）是由窗口系统产生的由某些操作触发的特殊函数，例如鼠标操作、键盘操作的一些事件，还有窗口显示、关闭、绘制等相关的事件。从 `QWidget` 继承的窗口部件常用的事件函数有如下几种。

- `closeEvent()`: 窗口关闭时触发的事件，通常在此事件做窗口关闭时的一些处理，例如显示一个对话框询问是否关闭窗口。
- `showEvent()`: 窗口显示时触发的事件。
- `paintEvent()`: 窗口绘制事件，第 8 章介绍绘图时会用到。
- `mouseMoveEvent()`: 鼠标移动事件。
- `mousePressEvent()`: 鼠标键按下事件。
- `mouseReleaseEvent()`: 鼠标键释放事件。
- `keyPressEvent()`: 键盘按键按下事件。
- `keyReleaseEvent()`: 键盘按键释放事件。

要利用某个事件进行一些处理，需要在窗口类里重定义事件函数并编写响应代码。在后面的例子中，将逐渐演示一些事件的用法。

在本例中，要利用对话框的 `closeEvent()` 事件，在类定义中声明了此事件的函数，其实现代码如下：

```
void QWDialogLocate::closeEvent(QCloseEvent *event)
{ //窗口关闭 event
    MainWindow *parWind = (MainWindow*)parentWidget(); //获取父窗口指针
    parWind->setActLocateEnable(true); //使能 actTab_Locate
    parWind->setDlgLocateNull(); //将窗口指针设置为 NULL
}
```

在 `closeEvent()` 事件里，调用主窗口的两个函数，将 `actTab_Locate` 重新使能，将主窗口内指向对话框的指针设置为 `NULL`。主窗口中这两个函数的实现代码如下：

```
void MainWindow::setActLocateEnable(bool enable)
{
    ui->actTab_Locate->setEnabled(enable);
}

void MainWindow::setDlgLocateNull()
{
    dlgLocate=NULL;
}
```

利用 `closeEvent()` 事件，可以询问窗口是否退出，例如为主窗口添加 `closeEvent()` 事件的处理，代码如下：

```
void MainWindow::closeEvent(QCloseEvent *event)
{ //窗口关闭时询问是否退出
    QMessageBox::StandardButton result=QMessageBox::question(this,
```

```

        "确认", "确定要退出本程序吗? ",
        QMessageBox::Yes|QMessageBox::No |QMessageBox::Cancel,
        QMessageBox::No);
    if (result==QMessageBox::Yes)
        event->accept();
    else
        event->ignore();
}

```

这样，主窗口关闭时就会出现一个询问对话框，如果不单击“**Yes**”按钮，程序就不关闭；否则应用程序结束。

### 6.2.5 利用信号与槽实现交互操作

前面设计的 QWDialogLocate 对话框与主窗口之间的交互采用互相引用的方式，实现起来比较复杂。另外一种实现方式就是利用 Qt 的信号与槽机制，设计相应的信号和槽，将信号与槽关联起来，在进行某个操作时发射信号，槽函数自动响应。

对 MainWindow 和 QWDialogLocate 稍作修改，采用信号与槽机制实现交互操作。

下面是 MainWindow 类定义中与此相关的定义，包括两个槽函数和一个信号。

```

class MainWindow : public QMainWindow
{
public slots:
    void    setACellText(int row, int column, QString &text); //设置单元格内容
    void    setActLocateEnable(bool enable); //设置 actTab_Locate 的 enabled 属性
signals:
    void    cellIndexChanged(int rowNo, int colNo); //当前单元格发生变化
};

```

两个槽函数是对话框操作主窗口时，主窗口作出的响应。信号是主窗口上 tableView 的当前单元格发生变化时发射的一个信号，以便对话框作出响应。

下面是两个槽函数的实现，以及 tableView 的 clicked() 信号的槽函数里发射自定义信号的代码，代码中都无须引用对话框对象。

```

void MainWindow::setACellText(int row, int column, QString &text)
{ //定位到单元格，并设置字符串
    QModelIndex index=theModel->index(row,column); //获取模型索引
    theSelection->clearSelection();
    theSelection->setCurrentIndex(index,QItemSelectionModel::Select);
    theModel->setData(index,text,Qt::DisplayRole); //设置单元格字符串
}
void MainWindow::setActLocateEnable(bool enable)
{ //设置 actTab_Locate 的 enabled 属性
    ui->actTab_Locate->setEnabled(enable);
}
void MainWindow::on_tableView_clicked(const QModelIndex &index)
{ //单击单元格时发射信号，传递单元格的行号、列号
    emit cellIndexChanged(index.row(),index.column());
}

```

在主窗口上，“定位单元格”按钮的响应代码与前面有较大的差别。

```

void MainWindow::on_actTab_Locate_triggered()
{//创建 StayOnTop 的对话框, 对话框关闭时自动删除
    QWDialogLocate *dlgLocate = new QWDialogLocate(this);
    dlgLocate->setAttribute(Qt::WA_DeleteOnClose);
    Qt::WindowFlags flags=dlgLocate->>windowFlags();
    dlgLocate->setWindowFlags(flags | Qt::WindowStaysOnTopHint);
    dlgLocate->setSpinRange(theModel->rowCount(),theModel->columnCount());
    QModelIndex curIndex=theSelection->currentIndex();
    if (curIndex.isValid())
        dlgLocate->setSpinValue(curIndex.row(),curIndex.column());
    //对话框发射信号, 设置单元格文字
    connect(dlgLocate,SIGNAL(changeCellText(int,int,QString&)),
            this,SLOT(setACellText(int,int,QString&)));
    //对话框发射信号, 设置 actTab_Locate 的属性
    connect(dlgLocate,SIGNAL(changeActionEnable(bool)),
            this,SLOT(setActLocateEnable(bool)));
    //主窗口发射信号, 修改对话框上的 spinBox 的值
    connect(this,SIGNAL(cellIndexChanged(int,int)),
            dlgLocate,SLOT(setSpinValue(int,int)));
    dlgLocate->show(); //非模态显示对话框
}

```

在这里, 对话框变量声明为了局部变量, 不再需要在主窗口类里保存对话框的指针。这段代码的关键是设置了 3 对信号与槽的关联。

在 QWDialogLocate 类定义中, 与信号和槽相关的定义如下。

```

class QWDialogLocate : public QDialog
{
private:
    void closeEvent(QCloseEvent *event);
    void showEvent(QShowEvent *event);
private slots:
    void on_btnSetText_clicked();
public slots:
    void setSpinValue(int rowNo, int colNo);
signals:
    void changeCellText(int row, int column, QString &text);
    void changeActionEnable(bool en);
};

```

QWDialogLocate 自定义了一个槽函数和两个信号, 还增加了 showEvent()事件的处理, 用于对话框显示时发射信号使主窗口的 actTab\_Locate 失效。这些槽函数, 以及发射信号的实现代码如下, 代码中没有出现对主窗口的引用。

```

void QWDialogLocate::closeEvent(QCloseEvent *event)
{ //窗口关闭事件, 发射信号使 actTab_Locate 能用
    emit changeActionEnable(true);
}
void QWDialogLocate::showEvent(QShowEvent *event)
{//窗口显示事件, 发射信号使 actTab_Locate 不能用
    emit changeActionEnable(false);
}
void QWDialogLocate::setSpinValue(int rowNo, int colNo)
{//响应主窗口信号, 更新 spinBox 的值
    ui->spinBoxRow->setValue(rowNo);
}

```



```

    ui->spinBoxColumn->setValue(colNo);
}
void QWDialogLocate::on_btnSetText_clicked()
{ //发射信号, 定位到单元格并设置字符串
    int row=ui->spinBoxRow->value(); //行号
    int col=ui->spinBoxColumn->value(); //列号
    QString text=ui->edtCaption->text(); //文字
    emit changeCellText(row,col,text); //发射信号
    if (ui->chkBoxRow->isChecked()) //行增
        ui->spinBoxRow->setValue(1+ui->spinBoxRow->value());
    if (ui->chkBoxColumn->isChecked()) //列增
        ui->spinBoxColumn->setValue(1+ui->spinBoxColumn->value());
}

```

经过这样修改后的程序, 能实现与前面的实例完全相同的主窗口与对话框交互的功能, 但是与前面互相引用的方式不同, 这里使用 Qt 的信号与槽的机制, 无须获取对方的指针, 程序结构上更简单一些。

## 6.3 多窗体应用程序设计

### 6.3.1 主要的窗体类及其用途

常用的窗体基类是 QWidget、QDialog 和 QMainWindow, 在创建 GUI 应用程序时选择窗体基类就是从这 3 个类中选择。QWidget 直接继承于 QObject, 是 QDialog 和 QMainWindow 的父类, 其他继承于 QWidget 的窗体类还有 QSplashScreen、QMdiSubWindow 和 QDesktopWidget。另外还有一个类 QWindow, 它同时从 QObject 和 QSurface 继承。这些类的继承关系如图 6-9 所示。

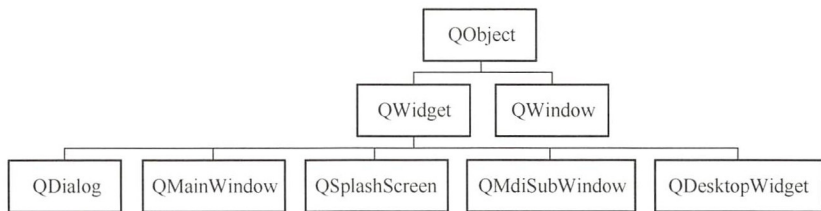


图 6-9 几个窗体类的继承关系

这些窗体类的主要特点和用途如下。

- **QWidget**: 在没有指定父容器时可作为独立的窗口, 指定父容器后可以作为容器的内部组件。
- **QDialog**: 用于设计对话框, 以独立窗口显示。
- **QMainWindow**: 用于设计带有菜单栏、工具栏、状态栏的主窗口, 一般以独立窗口显示。
- **QSplashScreen**: 用作应用程序启动时的 splash 窗口, 没有边框。
- **QMdiSubWindow**: 用于为 QMdiArea 提供一个子窗体, 用于 MDI (多文档) 应用程序的设计。
- **QDesktopWidget**: 具有多个显卡和多个显示器的系统具有多个桌面, 这个类提供用户桌面信息, 如屏幕个数、每个屏幕的大小等。

- QWindow: 通过底层的窗口系统表示一个窗口的类, 一般作为一个父容器的嵌入式窗体, 不作为独立窗体。

### 6.3.2 窗体类重要特性的设置

窗体显示或运行的一些特性可以通过 QWidget 的一些函数设置, 如 6.2 节介绍对话框的创建和使用时, 有如下的代码:

```
dlgLocate = new QWDialogLocate(this);
dlgLocate->setAttribute(Qt::WA_DeleteOnClose);
Qt::WindowFlags flags=dlgLocate->>windowFlags();
dlgLocate->setWindowFlags(flags | Qt::WindowStaysOnTopHint);
```

这段代码就用到了两个设置函数——setAttribute()和 setWindowFlags(), 它们可以设置窗体的显示特性和运行特性。下面介绍 QWidget 类中用于窗体属性设置的几个主要函数的功能。

#### 1. setAttribute()函数

setAttribute()函数用于设置窗体的一些属性, 其函数原型为:

```
void QWidget::setAttribute(Qt::WidgetAttribute attribute, bool on = true)
```

枚举类型 Qt::WidgetAttribute 定义了窗体的一些属性, 可以打开或关闭这些属性。枚举类型 Qt::WidgetAttribute 常用的常量及其意义见表 6-2。

表 6-2 枚举类型 Qt::WidgetAttribute 常用的常量

常量	意义
Qt::WA_AcceptDrops	允许窗体接收拖放来的组件
Qt::WA_DeleteOnClose	窗体关闭时删除自己, 释放内存
Qt::WA_Hover	鼠标进入或移出窗体时产生 paint 事件
Qt::WA_AcceptTouchEvents	窗体是否接受触屏事件

#### 2. setWindowFlags()函数

setWindowFlags()函数用于设置窗体标记, 其函数原型是:

```
void QWidget::setWindowFlags(Qt::WindowFlags type)
```

参数 type 是枚举类型 Qt::WindowType 的值的组合, 用于同时设置多个标记。

另外一个函数 setWindowFlag()用于一次设置一个标记, 其函数原型为:

```
void QWidget::setWindowFlag(Qt::WindowType flag, bool on = true)
```

可单独打开或关闭某个属性。枚举类型 Qt::WindowType 常用的常量值见表 6-3。

表 6-3 枚举类型 Qt::WindowType 常用的常量

常量	意义
表示窗体类型的常量	
Qt::Widget	这是 QWidget 类的缺省类型。这种类型的窗体, 如果它有父窗体, 就作为父窗体的子窗体; 否则就作为一个独立的窗口
Qt::Window	表明这个窗体是一个窗口, 通常具有窗口的边框、标题栏, 而不管它是否有父窗体
Qt::Dialog	表明这个窗体是一个窗口, 并且要显示为对话框 (例如在标题栏没有最小化、最大化按钮)。这是 QDialog 类的缺省类型

续表

常量	意义
Qt::Popup	表明这个窗体是用作弹出式菜单的窗体
Qt::Tool	表明这个窗体是工具窗体，具有更小的标题栏和关闭按钮，通常作为工具栏的窗体
Qt::ToolTip	表明这是用于 Tooltip 消息提示的窗体
Qt::SplashScreen	表明窗体是 splash 屏幕，是 QSplashScreen 类的缺省类型
Qt::Desktop	表明窗体是桌面，这是 QDesktopWidget 类的类型
Qt::SubWindow	表明窗体是子窗体，例如 QMdiSubWindow 就是这种类型
控制窗体显示效果的常量	
Qt::MSWindowsFixedSizeDialogHint	在 Windows 平台上，使窗口具有更窄的边框，用于固定大小的对话框
Qt::FramelessWindowHint	创建无边框窗口
定制窗体外观的常量，要定义窗体外观，需要先设置 Qt::CustomizeWindowHint	
Qt::CustomizeWindowHint	关闭缺省的窗口标题栏
Qt::WindowTitleHint	窗口有标题栏
Qt::WindowSystemMenuHint	有窗口系统菜单
Qt::WindowMinimizeButtonHint	有最小化按钮
Qt::WindowMaximizeButtonHint	有最大化按钮
Qt::WindowMinMaxButtonsHint	有最小化、最大化按钮
Qt::WindowCloseButtonHint	有关闭按钮
Qt::WindowContextHelpButtonHint	有上下文帮助按钮
Qt::WindowStaysOnTopHint	窗口总是处于最上层
Qt::WindowStaysOnBottomHint	窗口总是处于最下层
Qt::WindowTransparentForInput	窗口只作为输出，不接受输入

Qt::Widget、Qt::Window 等表示窗体类型的常量可以使窗体具有缺省的外观设置，如果设置为 Qt::Dialog 类型，则窗体具有对话框的缺省外观，例如标题栏没有最小化、最大化按钮。

控制窗体显示效果和外观的设置项可定制窗体的外观，例如设置一个窗体只有最小化最大化按钮，没有关闭按钮。

### 3. setWindowState() 函数

setWindowState() 函数使窗口处于最小化、最大化等状态，其函数原型是：

```
void QWidget::setWindowState(Qt::WindowStates windowState)
```

枚举类型 Qt::WindowState 表示了窗体的状态，其取值见表 6-4。

表 6-4 枚举类型 Qt::WindowState 的常量

常量	意义
Qt::WindowNoState	正常状态
Qt::WindowMinimized	窗口最小化
Qt::WindowMaximized	窗口最大化
Qt::WindowFullScreen	窗口填充整个屏幕，而且没有边框
Qt::WindowActive	变为活动的窗口，例如可以接收键盘输入

### 4. setWindowModality() 函数

setWindowModality() 函数用于设置窗口的模态，只对窗口类型有用。其函数原型为：

```
void setWindowModality(Qt::WindowModality windowModality)
```

枚举类型 Qt::WindowModality 的取值意义见表 6-5。

表 6-5 枚举类型 Qt:: WindowModality 的常量

常量	意义
Qt::NonModal	无模态，不会阻止其他窗口的输入
Qt::WindowModal	窗口对于其父窗口、所有的上级父窗口都是模态的
Qt::ApplicationModal	窗口对整个应用程序是模态的，阻止所有窗口的输入

### 5. setWindowOpacity()函数

setWindowOpacity()函数用于设置窗口的透明度，其函数原型如下：

```
void QWidget::setWindowOpacity(qreal level)
```

参数 level 是 1.0（完全不透明）至 0.0（完全透明）之间的数。窗口透明度缺省值是 1.0，即完全不透明。

## 6.3.3 多窗口应用程序的设计

### 1. 主窗口设计

本节以实例 samp6\_3 演示多窗口应用程序的设计，实例主窗口如图 6-10 所示。

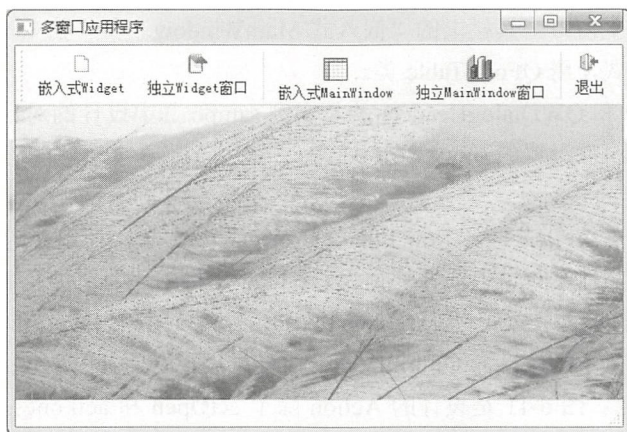


图 6-10 实例 samp6\_3 的主窗口

程序的主窗口类是 QMainWindow，从 QMainWindow 继承。主窗口有一个工具栏，4 个创建窗体的按钮以不同方式创建和使用窗体。主窗体工作区绘制一个背景图片，有一个 tabWidget 组件，作为创建窗体的父窗体。没有子窗体时，tabWidget 不显示。

下面是 QMainWindow 的构造函数和绘制背景图片的代码。

```
QMainWindow::QMainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::QMainWindow)
{
    ui->setupUi(this);
    ui->tabWidget->setVisible(false);
    ui->tabWidget->clear(); //清除所有页面
    ui->tabWidget->tabsClosable(); //Page 有关闭按钮，可被关闭
    this->setCentralWidget(ui->tabWidget);
    this->setWindowState(Qt::WindowMaximized); //窗口最大化显示
```



```

}
void QWMainWindow::paintEvent(QPaintEvent *event)
{ //绘制窗口背景图片
    Q_UNUSED(event);
    QPainter painter(this);
    painter.drawPixmap(0,ui->mainToolBar->height(),this->width(),
        this->height()-ui->mainToolBar->height()-ui->statusBar->height(),
        QPixmap(":/images/images/back.jpg"));
}

```

在构造函数中，将 `tabWidget` 组件设置为不可见，并且页面可关闭，这样每个页面标题部分都会出现一个关闭按钮，单击可以关闭页面。

背景图片绘制使用窗体的 `paintEvent()` 事件，获取主窗口的画笔之后，将资源文件里的一个图片绘制在主窗口的工作区。绘图的内容在第8章详细介绍。

实例除了主窗口之外，还有两个窗口和两个对话框。

- `QFormDoc`：是继承于 `QWidget` 可视化设计的窗体，主窗口工具栏上的“嵌入式 `Widget`”和“独立 `Widget` 窗口”按钮将以两种方式使用 `QFormDoc` 类。
- `QFormTable`：是继承于 `QMainWindow` 可视化设计的窗体，其界面功能与实例 `samp6_2` 的主窗口类似，主窗口工具栏上的“嵌入式 `MainWindow`”和“独立 `MainWindow` 窗口”按钮将以两种方式使用 `QFormTable` 类。
- `QWDialogSize` 和 `QWDialogHeaders` 就是实例 `samp6_2` 中设计的对话框类，由 `QFormTable` 调用进行表格组件设置。

## 2. `QFormDoc` 类的设计

在 Qt Creator 单击“File”→“New File or Project”菜单项，在出现的对话框里选择创建 Qt Designer Form Class，并且在向导中选择基类为 `QWidget`，将创建的新类命名为 `QFormDoc`。

在 `QFormDoc` 的窗口上只放置一个 `QPlainTextEdit` 组件。由于 `QFormDoc` 是从 `QWidget` 继承而来的，在 UI 设计器里不能直接为 `QFormDoc` 设计工具栏，但是可以创建 Action，然后在窗体创建时用代码创建工具栏。图 6-11 是设计的 Action 除了 `actOpen` 和 `actFont` 之外，其他编辑操作的 Action 都和 `QPlainTextEdit` 相关槽函数关联，`actClose` 与窗口的 `close()` 槽函数关联。

Name	Used	Text	Shortcut	Checkable	ToolTip
 <code>actOpen</code>	<input type="checkbox"/>	打开		<input type="checkbox"/>	打开文件
 <code>actCut</code>	<input type="checkbox"/>	剪切	Ctrl+X	<input type="checkbox"/>	剪切
 <code>actCopy</code>	<input type="checkbox"/>	复制	Ctrl+C	<input type="checkbox"/>	复制
 <code>actPaste</code>	<input type="checkbox"/>	粘贴	Ctrl+V	<input type="checkbox"/>	粘贴
 <code>actFont</code>	<input type="checkbox"/>	字体		<input type="checkbox"/>	设置字体
 <code>actClose</code>	<input type="checkbox"/>	关闭		<input type="checkbox"/>	关闭本窗口
 <code>actUndo</code>	<input type="checkbox"/>	撤销	Ctrl+Z	<input type="checkbox"/>	撤销编辑操作
 <code>actRedo</code>	<input type="checkbox"/>	重复		<input type="checkbox"/>	重复编辑操作

图 6-11 `QFormDoc` 窗口设计的 Action

`actOpen` 用于打开文件，`actFont` 用于设置文档字体，这些功能在前面的例子里都遇到过，不是本实例的重点，不再介绍其实现代码。

在 QFormDoc 的构造函数里用代码创建工具栏和布局，也可以在析构函数里增加一个消息显示的对话框，以便观察窗体是何时被删除的。代码如下：

```
QFormDoc::QFormDoc(QWidget *parent) :QWidget(parent),ui(new Ui::QFormDoc)
{
    ui->setupUi(this);
    QToolBar* locToolBar = new QToolBar("文档",this); //创建工具栏
    locToolBar->addAction(ui->actOpen);
    locToolBar->addAction(ui->actFont);
    locToolBar->addSeparator();
    locToolBar->addAction(ui->actCut);
    locToolBar->addAction(ui->actCopy);
    locToolBar->addAction(ui->actPaste);
    locToolBar->addAction(ui->actUndo);
    locToolBar->addAction(ui->actRedo);
    locToolBar->addSeparator();
    locToolBar->addAction(ui->actClose);
    locToolBar->setToolButtonStyle(Qt::ToolButtonTextBesideIcon);

    QVBoxLayout *Layout = new QVBoxLayout();
    Layout->addWidget(locToolBar); //设置工具栏和编辑器上下布局
    Layout->addWidget(ui->plainTextEdit);
    Layout->setContentsMargins(2,2,2,2); //减小边框的宽度
    Layout->setSpacing(2);
    this->setLayout(Layout); //设置布局
}
QFormDoc::~QFormDoc()
{
    QMessageBox::information(this, "消息", "QFormDoc 对象被删除和释放");
    delete ui;
}
```

### 3. QFormDoc 类的使用

主窗口工具栏上的“嵌入式 Widget”按钮的响应代码如下：

```
void QWMainWindow::on_actWidgetInsite_triggered()
{ //创建 QFormDoc 窗体，并在 tabWidget 中显示
    QFormDoc *formDoc = new QFormDoc(this);
    formDoc->setAttribute(Qt::WA_DeleteOnClose); //关闭时自动删除
    int cur=ui->tabWidget->addTab(formDoc,
        QString::asprintf("Doc %d",ui->tabWidget->count()));
    ui->tabWidget->setCurrentIndex(cur);
    ui->tabWidget->setVisible(true);
}
```

这段代码动态创建一个 QFormDoc 类对象 formDoc，并设置其为关闭时删除。然后使用 QTabWidget 的 addTab()函数，为主窗口上的 tabWidget 新建一个页面，作为 formDoc 的父窗体组件，formDoc 就在新建的页面里显示，我们称这种窗体显示方式为“嵌入式”。

主窗口工具栏上的“独立 Widget 窗口”按钮响应代码如下：

```
void QWMainWindow::on_actWidget_triggered()
{
    QFormDoc *formDoc = new QFormDoc();
    formDoc->setAttribute(Qt::WA_DeleteOnClose); //关闭时自动删除
```

```

formDoc->setWindowTitle("基于QWidget的窗体, 无父窗口, 关闭时删除");
formDoc->setWindowFlag(Qt::Window, true);
// formDoc->setWindowFlag(Qt::CustomizeWindowHint, true);
// formDoc->setWindowFlag(Qt::WindowMinMaxButtonsHint, false);
// formDoc->setWindowFlag(Qt::WindowCloseButtonHint, true);
// formDoc->setWindowFlag(Qt::WindowStaysOnTopHint, true);
formDoc->setWindowOpacity(0.9);
// formDoc->setWindowModality(Qt::WindowModal);
formDoc->show(); //在单独的窗口中显示
}

```

这里在创建 formDoc 对象时, 并没有指定父窗口, 创建窗口的代码是:

```
QFormDoc *formDoc = new QFormDoc();
```

使用 setWindowFlag()函数, 设置其为 Qt::Window 类型, 并用 show()函数显示窗口。这样创建的是一个单独显示的窗口, 并且在 windows 的任务栏上会有显示。若有文档窗口打开, 则关闭主窗口, 而文档窗口依然存在, 实际上这时候主窗口是隐藏了。若关闭所有文档窗口, 主窗口自动删除并释放, 才完全关闭应用程序。

如果创建 formDoc 时指定主窗口为父窗口, 即:

```
QFormDoc *formDoc = new QFormDoc(this);
```

则 formDoc 不会在 windows 的任务栏上显示, 关闭主窗口时, 所有文档窗口自动删除。

图 6-12 是嵌入式和独立的 QFormDoc 窗体的显示效果, 在创建独立的显示窗口时, 还可以尝试使用 setWindowFlag()函数设置不同的属性, 观察这些属性的控制效果。

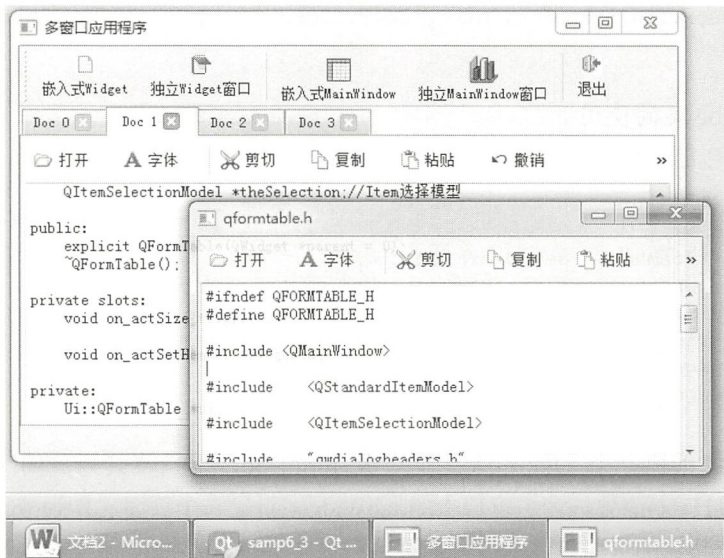


图 6-12 嵌入式和独立的 QFormDoc 窗体显示效果

#### 4. QFormTable 类的设计

表格窗口类 QFormTable 是基于 QMainWindow 的可视窗口类, 其功能与实例 samp6\_2 主窗口类似, 使用 QStandardItemModel 模型和 QTableView 组件构成 Model/View 结构的表格数据编辑器,

并且可以调用 `QWDialogSize` 和 `QWDialogHeaders` 对话框进行表格大小设置和表头设置。该窗口的具体设计不详细介绍了，只是为了观察窗口删除的时机，在析构函数里增加一个信息显示对话框。

```
QFormTable::~QFormTable()
{
    QMessageBox::information(this, "消息", "FormTable 窗口被删除和释放");
    delete ui;
}
```

## 5. QFormTable 类的使用

主窗口工具栏上的“嵌入式 MainWindow”按钮的响应代码如下：

```
void QWMainWindow::on_actWindowInsite_triggered()
{
    QFormTable *formTable = new QFormTable(this);
    formTable->setAttribute(Qt::WA_DeleteOnClose); //关闭时自动删除
    int cur=ui->tabWidget->addTab(formTable,
        QString::asprintf("Table %d",ui->tabWidget->count()));
    ui->tabWidget->setCurrentIndex(cur);
    ui->tabWidget->setVisible(true);
}
```

代码功能是创建一个 `QFormTable` 对象 `formTable`，并在主窗口的 `tabWidget` 组件里新增一个页面，将 `formTable` 显示在新增页面里。所以，即使是从 `QMainWindow` 继承的窗口类，也是可以在其他界面组件里嵌入式显示的。

主窗口工具栏上的“独立 MainWindow 窗口”按钮响应代码如下：

```
void QWMainWindow::on_actWindow_triggered()
{
    QFormTable* formTable = new QFormTable(this);
    formTable->setAttribute(Qt::WA_DeleteOnClose);
    formTable->setWindowTitle("基于 QMainWindow 的窗口，指定父窗口，关闭时删除");
    formTable->show();
}
```

这样创建的 `formTable` 以独立窗口显示，关闭时自动删除。它指定了主窗口为父窗口，主窗口关闭时，所有 `QFormTable` 类窗口自动删除。

无论是嵌入式的，还是独立的 `QFormTable` 窗口，都可以调用 `QWDialogSize` 和 `QWDialogHeaders` 对话框进行表格大小和表头文字设置，对话框的调用方法在 6.2 节已有介绍。创建 `QFormTable` 嵌入式窗体和独立窗口的运行效果如图 6-13 所示。

## 6. QTabWidget 类的控制

现在，单击 `tabWidget` 中嵌入的 `QFormDoc` 或 `QFormTable` 窗体工具栏上的“关闭”按钮，都可以关闭窗体并且删除分页。但是单击分页上的关闭图标，并不能关闭窗口。而且，关闭所有分页后，`tabWidget` 并没有隐藏，无法显示背景图片。

为此，需要对 `tabWidget` 的两个信号编写槽函数，`tabCloseRequested()`和 `currentChanged()`信号的槽函数代码如下：

```
void QWMainWindow::on_tabWidget_tabCloseRequested(int index)
{ //关闭 Tab
    if (index<0)
        return;
```



```

QWidget* aForm=ui->tabWidget->widget(index);
aForm->close();
}
void QMainWindow::on_tabWidget_currentChanged(int index)
{
    bool en=ui->tabWidget->count()>0; //再无分页
    ui->tabWidget->setVisible(en);
}

```

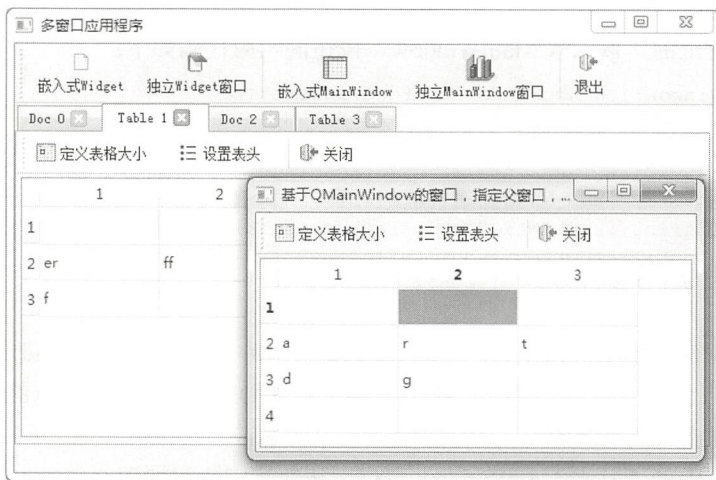


图 6-13 嵌入式和独立的 QFormDoc 窗体显示效果

tabCloseRequested()信号在单击分页的关闭图标时发射,传递来的参数 index 表示页面的编号。QTabWidget::widget()返回 TabWidget 组件中某个页面的窗体组件。获取页面的 QWidget 组件后,调用 close()函数关闭窗体。

删除一个分页或切换页面时,会发射 currentChanged()信号,在此信号的槽函数里判断分页个数是否为零,以控制 tabWidget 是否可见。

## 6.4 MDI 应用程序设计

### 6.4.1 MDI 简介

传统的应用程序设计中有多文档界面 (Multi-document Interface, MDI) 应用程序,Qt 为设计 MDI 应用程序提供了支持。

本节的实例 samp6\_4 是一个 MDI 应用程序,程序运行效果如图 6-14 所示。MDI 应用程序就是在主窗口里创建多个同类型的 MDI 子窗口,这些 MDI 子窗口在主窗口里显示,并共享主窗口上的工具栏和菜单等操作功能,主窗口上的操作都针对当前活动的 MDI 子窗口进行。

设计 MDI 应用程序需要在主窗口工作区放置一个 QMdiArea 作为子窗体的容器。实例 samp6\_4 主窗口的工作区使用一个 QMdiArea 组件,实例的子窗口类是 QFormDoc,是一个使用 QPlainTextEdit 进行简单文本显示和编辑的窗体。



图 6-14 MDI 应用程序实例 samp6\_4 的运行时界面

创建的 QFormDoc 窗体对象作为一个子窗口加入到 mdiArea 组件中。QMdiArea 组件类似于实例 samp6\_3 中主窗口上的 tabWidget 组件，只是 QMdiArea 提供更加完备的功能。更改 MDI 的显示模式，可以得到与实例 samp6\_3 相似的以多页组件管理的 MDI 界面效果。

### 6.4.2 文档窗口类 QFormDoc 的设计

以可视化方式创建一个基于 QWidget 的类 QFormDoc，设计可视化界面时，只放置一个 QPlainTextEdit 组件，并以水平布局填充整个窗口。这里不再用可视化的方式设计 Action，因为 QFormDoc 窗口不需要创建自己的工具栏，而是使用主窗口上的工具栏按钮对 QFormDoc 窗体上的 QPlainTextEdit 组件进行操作。

为 QFormDoc 添加一些用于文件打开和编辑操作的接口函数，QFormDoc 类的完整定义如下：

```
class QFormDoc : public QWidget
{
    Q_OBJECT
private:
    QString mCurrentFile; //当前文件
    bool mFileOpened=false; //文件已打开
public:
    explicit QFormDoc(QWidget *parent = 0);
    ~QFormDoc();
    void loadFromFile(QString& aFileName); //打开文件
    QString currentFileName(); //返回当前文件名
    bool isFileOpened(); //文件已经打开
    void setEditFont(); //设置字体
    void textCut(); //cut
    void textCopy(); //copy
    void textPaste(); //paste
private:
    Ui::QFormDoc *ui;
};
```

这些接口函数是为了在主窗口里调用,实现对 MDI 子窗口的操作。实现代码如下:

```
QFormDoc::QFormDoc(QWidget *parent) : QWidget(parent), ui(new Ui::QFormDoc)
{
    ui->setupUi(this);
    this->setWindowTitle("New Doc"); //窗口标题
    this->setAttribute(Qt::WA_DeleteOnClose); //关闭时自动删除
}
QFormDoc::~QFormDoc()
{
    QMessageBox::information(this, "信息", "文档窗口被释放");
    delete ui;
}
void QFormDoc::loadFromFile(QString &aFileName)
{//打开文件
    QFile aFile(aFileName);
    if (aFile.open(QIODevice::ReadOnly | QIODevice::Text))
    {
        QTextStream aStream(&aFile); //用文本流读取文件
        ui->plainTextEdit->clear();
        ui->plainTextEdit->setPlainText(aStream.readAll()); //读取文本文件
        aFile.close(); //关闭文件
        mCurrentFile=aFileName; //保存当前文件名
        QFileInfo fileInfo(aFileName); //文件信息
        QString str=fileInfo.fileName(); //去除路径后的文件名
        this->setWindowTitle(str);
        mFileOpened=true;
    }
}
QString QFormDoc::currentFileName()
{ return mCurrentFile;
}
bool QFormDoc::isFileOpened()
{ return mFileOpened;
}
void QFormDoc::setEditFont()
{
    QFont font=ui->plainTextEdit->font();
    bool ok;
    font=QFontDialog::getFont(&ok,font);
    ui->plainTextEdit->setFont(font);
}
void QFormDoc::textCut()
{ ui->plainTextEdit->cut();
}
void QFormDoc::textCopy()
{ ui->plainTextEdit->copy();
}
void QFormDoc::textPaste()
{ ui->plainTextEdit->paste();
}
```

**注意** 作为 MDI 子窗口,不管其是否设置为关闭时删除,在主窗口里关闭一个 MDI 子窗口时,都会删除子窗口对象。

### 6.4.3 MDI 主窗口设计与子窗口的使用

#### 1. 主窗口界面设计

要在主窗口实现 MDI 功能,只需在主窗口的工作区放置一个 QMdiArea 组件。图 6-15 是设计好的主窗口界面。

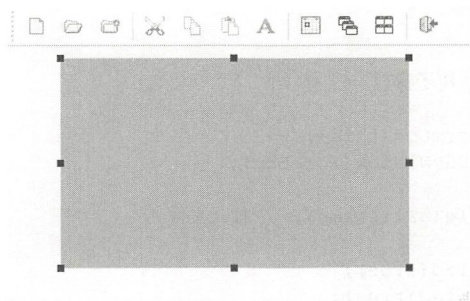


图 6-15 设计时的主窗口

在 UI 设计器里创建 Action,并应用 Action 设计主工具栏。在主窗口的工作区放置一个 QMdiArea 组件,然后在主窗口的构造函数里设置 mdiArea 填满工作区。

```
QWMainWindow::QWMainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::QWMainWindow)
{
    ui->setupUi(this);
    this->setCentralWidget(ui->mdiArea);
    this->setWindowState(Qt::WindowMaximized);
    ui->mainToolBar->setToolButtonStyle(Qt::ToolButtonTextUnderIcon);
}
```

#### 2. MDI 子窗口的创建与加入

下面是主窗口上“新建文档”按钮的响应代码:

```
void QWMainWindow::on_actDoc_New_triggered()
{ //新建文档
    QFormDoc *formDoc = new QFormDoc(this);
    ui->mdiArea->addSubWindow(formDoc); //文档窗口添加到 MDI
    formDoc->show();
}
```

代码功能是新建一个 QFormDoc 类的窗口 formDoc,构造函数中传入了主窗口指针,所以主窗口是 formDoc 的父窗口,然后使用 QMdiArea 的 addSubWindow()函数将 formDoc 加入到 mdiArea。

下面是主窗口上“打开文档”按钮的响应代码:

```
void QWMainWindow::on_actDoc_Open_triggered()
{ //打开文件
    bool needNew=false; // 是否需要新建子窗口
    QFormDoc *formDoc;
    if (ui->mdiArea->subWindowList().count()>0) //获取活动窗口
    {
        formDoc=(QFormDoc*)ui->mdiArea->activeSubWindow()->widget();
        needNew=formDoc->isFileOpened();//文件已经打开,需要新建窗口
    }
}
```



```

    }
    else
        needNew=true;

    QString curPath=QDir::currentPath();
    QString aFileName=QFileDialog::getOpenFileName(this,"打开一个文件",
        curPath, "C 程序文件(*.h *.cpp);;所有文件 (*.*)");
    if (aFileName.isEmpty())
        return;

    if (needNew) //需要新建子窗口
    {
        formDoc = new QFormDoc(this);
        ui->mdiArea->addSubWindow(formDoc);
    }
    formDoc->loadFromFile(aFileName); //打开文件
    formDoc->show();
    ui->actCut->setEnabled(true);
    ui->actCopy->setEnabled(true);
    ui->actPaste->setEnabled(true);
    ui->actFont->setEnabled(true);
}

```

通过 `QMdiArea::subWindowList()` 可以获得子窗口对象列表，从而可以判断子窗口的个数。如果没有一个 MDI 子窗口，就创建一个新的窗口并打开文件。

如果有 MDI 子窗口，则总有一个活动窗口，通过 `QMdiArea::activeSubWindow()` 可以获得此活动的子窗口，通过子窗口的 `isFileOpened()` 函数判断是否打开了文件，如果没有打开过文件，就在这个活动窗口里打开文件，否则新建窗口打开文件。

---

**注意** 一定要先获取 MDI 子窗口，再使用 `QFileDialog` 选择需要打开的文件。如果顺序更换了，则无法获得正确的 MDI 活动子窗口。

---

### 3. QMdiArea 常用功能函数

`QMdiArea` 提供了一些成员函数，可以进行一些操作，工具栏上的“关闭全部”“MDI 模式”“级联展开”“平铺展开”等按钮都是调用 `QMdiArea` 类的成员函数实现的。下面是这几个按钮功能的实现代码：

```

void QWMainWindow::on_actCascade_triggered()
{ //窗口级联展开
    ui->mdiArea->cascadeSubWindows();
}
void QWMainWindow::on_actTile_triggered()
{ //平铺展开
    ui->mdiArea->tileSubWindows();
}
void QWMainWindow::on_actCloseALL_triggered()
{ //关闭全部子窗口
    ui->mdiArea->closeAllSubWindows();
}
void QWMainWindow::on_actViewMode_triggered(bool checked)
{ //MDI 显示模式

```

```

if (checked) //Tab 多页显示模式
{
    ui->mdiArea->setViewMode(QMdiArea::TabbedView); //Tab 多页显示模式
    ui->mdiArea->setTabsClosable(true); //页面可关闭
    ui->actCascade->setEnabled(false);
    ui->actTile->setEnabled(false);
}
else //子窗口模式
{
    ui->mdiArea->setViewMode(QMdiArea::SubWindowView); //子窗口模式
    ui->actCascade->setEnabled(true);
    ui->actTile->setEnabled(true);
}
}

```

其中, 设置 MDI 视图模式用 `setViewMode()` 函数, 有两种模式可以选择。

- `QMdiArea::SubWindowView` 是传统的子窗口模式, 显示效果如图 6-14 所示。
- `QMdiArea::TabbedView` 是多页的显示模式, 显示效果如图 6-16 所示。

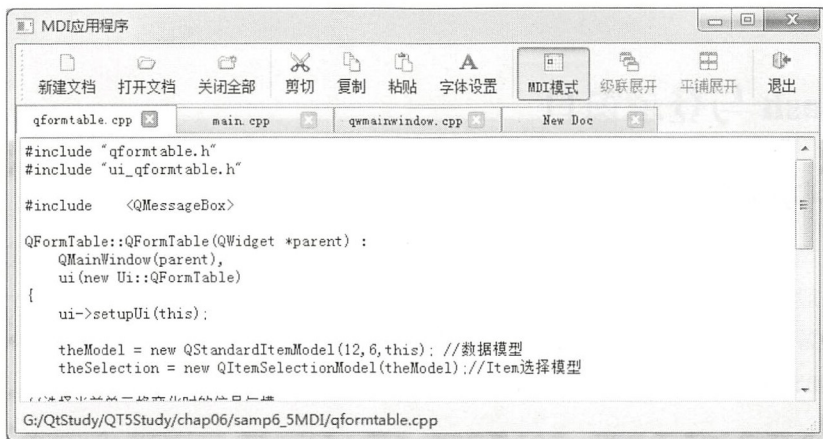


图 6-16 多页模式下的 MDI 界面

#### 4. MDI 的信号

`QMdiArea` 有一个信号 `subWindowActivated(QMdiSubWindow *arg1)`, 在当前活动窗口切换时发射, 利用此信号可以在活动窗口切换时进行一些处理, 例如, 在状态栏里显示活动 MDI 子窗口的文件名, 在没有 MDI 子窗口时, 将工具栏上的编辑功能按钮设置为禁用。下面是该信号的槽函数代码:

```

void QMdiArea::on_mdiArea_subWindowActivated(QMdiSubWindow *arg1)
{
    //当前活动子窗口切换时
    if (ui->mdiArea->subWindowList().count() == 0)
    {
        //若子窗口个数为零
        ui->actCut->setEnabled(false);
        ui->actCopy->setEnabled(false);
        ui->actPaste->setEnabled(false);
        ui->actFont->setEnabled(false);
        ui->statusBar->clearMessage();
    }
}

```

```

else
{
    QFormDoc *formDoc=static_cast<QFormDoc*>(
        ui->mdiArea->activeSubWindow()->widget());
    ui->statusBar->showMessage(formDoc->currentFileName());
}
}

```

主窗口工具栏上的“剪切”“复制”“粘贴”“字体设置”等按钮都是调用当前子窗口的相应函数，关键是获取当前 MDI 子窗体对象，例如，“剪切”和“字体设置”按钮的代码如下：

```

void QWMainWindow::on_actCut_triggered()
{ //cut 操作
    QFormDoc* formDoc=(QFormDoc*)ui->mdiArea->activeSubWindow()->widget();
    formDoc->textCut();
}
void QWMainWindow::on_actFont_triggered()
{ //设置字体
    QFormDoc* formDoc=(QFormDoc*)ui->mdiArea->activeSubWindow()->widget();
    formDoc->setEditFont();
}

```

## 6.5 Splash 与登录窗口

### 6.5.1 实例功能概述

一般的大型应用程序在启动时会显示一个启动画面，即 Splash 窗口。Splash 窗口是一个无边对话框，一般显示一个图片，展示软件的信息。Splash 窗口显示时，程序在后台做一些比较耗时的启动准备工作，Splash 窗口显示一段时间后自动关闭，然后软件的主窗口显示出来。Qt 有一个 QSplashScreen 类可以实现 Splash 窗口的功能，它提供了载入图片，自动设置窗口无边框效果等功能。

有的应用程序还有软件登录界面，要求用户输入用户名和密码才可以进入软件。

Splash 窗口和登录界面实质都是对话框，它们在程序启动时显示。本节设计的实例 samp6\_5，是在实例 samp6\_4 基础上增加了一个 Splash 登录对话框，这个对话框结合了 Splash 窗口和登录界面两者的功能，实例运行时的启动界面如图 6-17 所示。

这个实例演示如下的一些功能的实现方法：

- 如何实现 Splash 特点的无边框对话框；
- 如何设计用鼠标拖动无边框的对话框；
- 如何使用 QSettings 类存储用户名、密码等信息；
- 如何用 QCryptographicHash 类进行字符串加密；



图 6-17 实例 samp6\_5 的 Splash 和登录窗口

- 如何根据登录输入状况确定启动主窗口或终止程序运行。

## 6.5.2 对话框界面设计和类定义

采用新建 Qt Designer Form Class 的方法创建启动登录对话框，它从 QDialog 继承而来，设置类名称为 QDlgLogin。界面设计在 UI 设计器里进行，主要区域是一个用于显示图片的 QLabel 组件，在资源文件里载入图片，为 QLabel 组件的 pixmap 指定图片。

对话框下方是用于用户名和密码输入的 QLineEdit 组件，两个按钮用于选择用户输入，设置“取消”按钮的 clicked() 信号与对话框的 reject() 槽函数关联。但是“确定”按钮的 clicked() 信号不要设置为与对话框的任何槽函数关联，需要对其编写自定义的槽函数代码，因为需要根据用户输入确定对话框返回结果。为对话框界面上的组件设置好布局。

下面是 qdlglogin.h 文件中 QDlgLogin 类的定义：

```
class QDlgLogin : public QDialog
{
    Q_OBJECT
private:
    bool    m_moving=false; //表示窗口是否在鼠标操作下移动
    QPoint  m_lastPos;    //上一次的鼠标位置
    QString m_user="user"; //初始化用户名
    QString m_pswd="12345"; //初始化密码，未加密的
    int     m_tryCount=0; //试错次数
    void    readSettings(); //读取设置,注册表
    void    writeSettings(); //写入设置,注册表
    QString encrypt(const QString& str); //字符串加密
protected:
    //用于鼠标拖动窗口的鼠标事件
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
public:
    explicit QDlgLogin(QWidget *parent = 0);
    ~QDlgLogin();
private slots:
    void on_btnOK_clicked();
private:
    Ui::dlgLogin *ui;
};
```

在 QDlgLogin 类中，定义了一些私有成员变量。

- m\_moving 和 m\_lastPos 用于在拖动窗口时记录移动状态和上次的位置，由于 Splash 窗口没有标题栏，所以采用在图片上拖拉的方式移动窗口，使用了 3 个鼠标事件来实现窗口拖动操作。
- m\_user, m\_pswd, m\_tryCount 用于记录用户名、密码和试错次数。
- readSettings() 用于读取存储的设置，writeSettings() 用于将设置存储，在 Windows 系统下，这些信息是存储在注册表里的。
- encrypt() 函数用于对一个字符串进行加密。



### 6.5.3 QDlgLogin 类功能实现

#### 1. 构造函数里的初始化

QDlgLogin 类的构造函数代码如下:

```
QDlgLogin::QDlgLogin(QWidget *parent) : QDialog(parent), ui(new Ui::dlgLogin)
{
    ui->setupUi(this);
    ui->editPSWD->setEchoMode(QLineEdit::Password); //设置为密码输入模式
    this->setAttribute(Qt::WA_DeleteOnClose); //设置为关闭时删除
    this->setWindowFlags(Qt::SplashScreen); //设置为 SplashScreen
    // this->setWindowFlags(Qt::FramelessWindowHint); //无边框, 但在任务栏显示标题
    readSettings(); //读取存储的用户名和密码
}
```

QLineEdit::setEchoMode()函数设置编辑框回显方式, 参数为 QLineEdit::EchoMode 枚举类型, 这里设置为 QLineEdit::Password 回显方式, 用于将密码输入回显为一个符号, 而不显示真实字符。

使用 setWindowFlags()函数将窗口标志设置为 Qt::SplashScreen, 这样对话框显示为 Splash 窗口, 无边框, 且在 Windows 任务栏上没有显示。另外一个类似的标志是 Qt::FramelessWindowHint, 它会使对话框无边框, 但是会在任务栏上显示对话框的标题。

初始设置后调用 readSettings()函数读取存储的设置, 根据存储的情况将用户名显示到窗口上的编辑框里。

#### 2. 应用程序设置的存储

自定义成员函数 readSettings()用于读取应用程序设置, writeSettings()用于保存设置, 实现代码如下:

```
void QDlgLogin::readSettings()
{ //读取存储的用户名和密码, 密码是经过加密的
    QString organization="WWB-Qt"; //用于注册表,
    QString appName="samp6_5";
    QSettings settings(organization, appName);
    bool saved=settings.value("saved", false).toBool(); //读取 saved
    m_user=settings.value("Username", "user").toString(); //读取 Username
    QString defaultPSWD=encrypt("12345"); //缺省密码"12345"加密后的数据
    m_pswd=settings.value("PSWD", defaultPSWD).toString(); //读取 PSWD
    if (saved)
        ui->editUser->setText(m_user);
    ui->checkBoxSave->setChecked(saved);
}

void QDlgLogin::writeSettings()
{ //保存用户名, 密码等设置
    QSettings settings("WWB-Qt", "samp6_5"); //注册表键组
    settings.setValue("Username", m_user); //用户名
    settings.setValue("PSWD", m_pswd); //密码, 经过加密的
    settings.setValue("saved", ui->checkBoxSave->isChecked());
}
```

应用程序的设置是指应用程序需要保存的一些信息, 在 Windows 系统下, 这些信息保存在注册表里。使用 QSettings 类可以实现设置信息的读取和写入。

创建 QSettings 对象时, 需要传递 organization 和 appName, 例如:

```
QSettings settings("WWB-Qt", "samp6_5");
```

指向的注册表目录是 HKEY\_CURRENT\_USER/Software/WWB-Qt/samp6\_5。

注册表里参数是以“键——键值”对来保存的。writeSettings()函数里使用 setValue()函数写入键值, readSettings()里使用 value()函数读取键值。读取键值时可以指定缺省值, 即如果键不存在, 就用缺省值作为读取的值。

在 Windows 的开始菜单的输入框里输入 regedit, 打开注册表, 查找到目录 HKEY\_CURRENT\_USER/Software/WWB-Qt/samp6\_5, 可以看到注册表里参数存储情况。其中, 存储的密码是加密后的字符串。

### 3. 字符串加密

本实例中密码采用加密后的字符串保存, 这样在实际应用中具有安全性。Qt 提供了用于加密的类 QCryptographicHash, 自定义函数 encrypt()就利用这个类进行字符串加密, 实现代码如下:

```
QString QDlgLogin::encrypt(const QString &str)
{ //字符串 MD5 算法加密
    QByteArray btArray;
    btArray.append(str);
    QCryptographicHash hash(QCryptographicHash::Md5); //Md5 加密算法
    hash.addData(btArray); //添加数据
    QByteArray resultArray = hash.result(); //返回最终的散列值
    QString md5 = resultArray.toHex(); //转换为 16 进制字符串
    return md5;
}
```

QCryptographicHash 创建时需要指定一种加密算法, 加密算法变量是枚举类型 QCryptographicHash::Algorithm, 常用的常量值有 QCryptographicHash::Md4、QCryptographicHash::Md5、QCryptographicHash::Sha512 等, 完整的描述可参考 Qt 的帮助文档。

QCryptographicHash 只提供了加密功能, 没有提供解密功能。

### 4. 用户名和密码输入判断

登录窗口运行后, 单击“确定”按钮, 程序会对输入内容进行判断, “确定”按钮的槽函数代码如下:

```
void QDlgLogin::on_btnOK_clicked()
{ // "确定"按钮
    QString user = ui->editUser->text().trimmed(); //输入用户名
    QString pswd = ui->editPSWD->text().trimmed(); //输入密码
    QString encrptPSWD = encrypt(pswd); //对输入密码进行加密
    if ((user == m_user) && (encrptPSWD == m_pswd))
    {
        writeSettings();
        this->accept(); //对话框 accept(), 关闭对话框
    }
    else
    {
        m_tryCount++; //错误次数
        if (m_tryCount > 3)
        {
            QMessageBox::critical(this, "错误", "输入错误次数太多, 强行退出");
            this->reject(); //退出
        }
    }
}
```

```

        else
            QMessageBox::warning(this, "错误提示", "用户名或密码错误");
    }
}

```

由于 `QCryptographicHash` 只提供了加密功能，没有提供解密功能，所以，在读取应用程序设定后，无法将加密后的密码解密并显示在窗口上，程序只能回显用户名，而不能回显密码。

这段程序会对输入的密码进行加密，因为从注册表读取的是加密后的密码，所以能够对比输入的用户名和密码与存储的用户名和密码是否匹配。

如果输入正确，调用窗口的 `accept()` 槽函数关闭对话框，对话框返回值为 `QDialog::Accepted`，否则试错次数加一；如果试错次数大于 3 次，就调用窗口的 `reject()` 槽函数关闭对话框，对话框返回值为 `QDialog::Rejected`。

### 5. 窗口拖动功能的实现

由于 `Splash` 窗口没有边框，因此不能像普通的窗口那样通过拖动窗口的标题栏来拖动窗口。为了实现窗口的拖动功能，对窗口的 3 个鼠标事件进行处理，实现的代码如下：

```

void QDlgLogin::mousePressEvent(QMouseEvent *event)
{ //鼠标按键被按下
    if (event->button() == Qt::LeftButton)
    {
        m_moving = true;
        m_lastPos = event->globalPos() - pos(); //记录下鼠标相对于窗口的位置
    }
    return QDialog::mousePressEvent(event);
}

void QDlgLogin::mouseMoveEvent(QMouseEvent *event)
{ //鼠标按下左键移动
    if (m_moving && (event->buttons() && Qt::LeftButton) &&
        (event->globalPos() - m_lastPos).manhattanLength() > QApplication::startDragDistance())
    {
        move(event->globalPos() - m_lastPos);
        m_lastPos = event->globalPos() - pos();
    }
    return QDialog::mouseMoveEvent(event);
}

void QDlgLogin::mouseReleaseEvent(QMouseEvent *event)
{ //鼠标按键释放
    m_moving = false; //停止移动
}

```

`mousePressEvent(QMouseEvent *event)` 事件在鼠标按键按下时发生，传递的参数 `event` 有鼠标按键和坐标信息，判断如果是鼠标左键按下，就设置变量 `m_moving` 值为 `true`，表示开始移动，并记录下鼠标坐标。`event->globalPos()` 与对话框的 `pos()` 是不同坐标系下的坐标，在绘图这一章再详细介绍。

`mouseMoveEvent(QMouseEvent *event)` 事件在鼠标移动时发射，程序里判断是否已经开始移动并且按下鼠标左键；如果是，则调用窗口的 `move()` 函数，横向和纵向移动一定的距离，并再次记录坐标点。

`mouseReleaseEvent(QMouseEvent *event)` 事件在鼠标按键释放时发生，左键释放时停止窗口移动。

所以，当在窗口上按下鼠标左键并移动时，窗口就会随之移动。

### 6.5.4 Splash 登录窗口的使用

设计好启动和登录窗口 `QDlgLogin` 之后，在 `main()` 函数里使用启动与登录对话框。`main()` 函数的代码如下：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QDlgLogin *dlgLogin=new QDlgLogin;
    if (dlgLogin->exec()==QDialog::Accepted)
    {
        QWMainWindow w;
        w.show();
        return a.exec();
    }
    else
        return 0;
}
```

在主窗口之前创建 Splash 登录对话框对象 `dlgLogin`，并以模态显示的方式调用此对话框。如果对话框返回的是 `QDialog::Accepted`，说明通过了用户名和密码验证，就创建主窗口并显示；否则退出应用程序。由于 `QDlgLogin` 设置为关闭时删除，验证关闭登录窗口后，对象会自动删除。



## 第7章

# 文件系统和文件读写

文件的读写是很多应用程序具有的功能，甚至某些应用程序就是围绕着某一种格式文件的处理而开发的，所以文件读写是应用程序开发的一个基本功能。本章介绍 Qt 中如何实现文本文件、二进制文件的读写，以及文件和目录的管理功能。

## 7.1 文本文件读写

### 7.1.1 实例功能概述

文本文件是指以纯文本格式存储的文件，例如用 Qt Creator 编写的 C++ 程序的头文件（.h 文件）和源程序文件（.cpp 文件）。HTML 和 XML 文件也是纯文本文件，只是其读取之后需要对内容进行解析之后再显示。

Qt 提供了两种读写纯文本文件的基本方法，一种是用 QFile 类的 IODevice 读写功能直接进行读写，另一种是利用 QFile 和 QTextStream 结合起来，用流（Stream）的方法进行文件读写。

实例 samp7\_1 演示了这两种方法读写文本文件，其运行时窗口如图 7-1 所示。实例不仅演示了如何打开文本文件，还有文件保存功能。

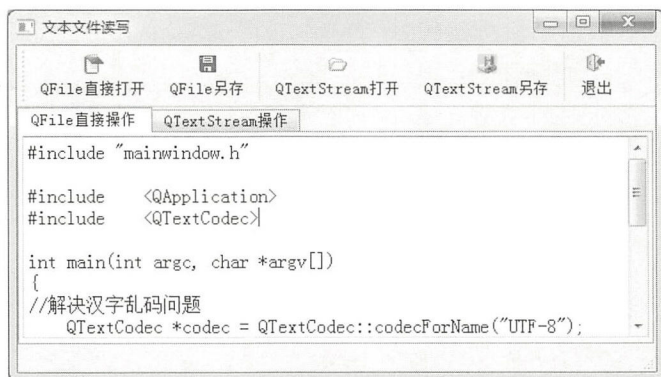


图 7-1 实例 samp7\_1 的运行时窗口

### 7.1.2 QFile 读写文本文件

QFile 类是直接于 IO 设备打交道，进行文件读写操作的类，使用 QFile 可以直接打开或保存

文本文件。图 7-1 工具栏上的“QFile 直接打开”按钮用 QFile 类的功能直接打开文本文件，按钮的槽函数及相关函数的代码如下：

```
void MainWindow::on_actOpen_IODevice_triggered()
{
    //打开文件
    QString curPath=QDir::currentPath();
    QString dlgTitle="打开一个文件";
    QString filter="程序文件 (*.h *.cpp);;文本文件 (*.txt);;所有文件 (*.*)";
    QString aFileName=QFileDialog::getOpenFileName(this,
        dlgTitle,curPath,filter);
    if (aFileName.isEmpty())
        return;
    openTextByIODevice(aFileName);
}

bool MainWindow::openTextByIODevice(const QString &aFileName)
{
    //用 IODevice 方式打开文本文件
    QFile aFile(aFileName);
    if (!aFile.exists()) //文件不存在
        return false;
    if (!aFile.open(QIODevice::ReadOnly | QIODevice::Text))
        return false;
    ui->textEditDevice->setPlainText(aFile.readAll());
    aFile.close();
    ui->tabWidget->setCurrentIndex(0);
    return true;
}
```

自定义函数 openTextByIODevice()实现文本文件打开的功能。定义 QFile 对象变量 aFile 时将文件名传递给它，检查文件存在后，通过 open()函数打开文件。

QFile::open()函数打开文件时需要传递 QIODevice::OpenModeFlag 枚举类型的参数，决定文件以什么方式打开，QIODevice::OpenModeFlag 类型的主要取值如下。

- QIODevice::ReadOnly：以只读方式打开文件，用于载入文件。
- QIODevice::WriteOnly：以只写方式打开文件，用于保存文件。
- QIODevice::ReadWrite：以读写方式打开。
- QIODevice::Append：以添加模式打开，新写入文件的数据添加到文件尾部。
- QIODevice::Truncate：以截取方式打开文件，文件原有的内容全部被删除。
- QIODevice::Text：以文本方式打开文件，读取时“\n”被自动翻译为换行符，写入时字符串结束符会自动翻译为系统平台的编码，如 Windows 平台下是“\r\n”。

这些取值可以组合，例如 QIODevice::ReadOnly | QIODevice::Text 表示以只读和文本方式打开文件。

将文件内容全部读出并设置为 QPlainTextEdit 组件的内容只需一条语句：

```
ui->textEditDevice->setPlainText(aFile.readAll());
```

文件内容读取结束后，需要调用 QFile::close()函数关闭文件。

图 7-1 工具栏上的“QFile 另存”按钮用 QFile 类的功能将 QPlainTextEdit 组件中的文本保存为一个文本文件，实现代码如下：

```

void MainWindow::on_actSave_IODevice_triggered()
{
    QString curPath=QDir::currentPath();
    QString dlgTitle="另存为一个文件";
    QString filter="h 文件 (*.h);;c++文件 (*.cpp);;所有文件 (*.*)";
    QString aFileName=QFileDialog::getSaveFileName(
        this,dlgTitle,curPath,filter);
    if (aFileName.isEmpty())
        return;
    saveTextByIODevice(aFileName);
}

bool MainWindow::saveTextByIODevice(const QString &aFileName)
{ //用 IODevice 方式保存文本文件
    QFile aFile(aFileName);
    if (!aFile.open(QIODevice::WriteOnly | QIODevice::Text))
        return false;
    QString str=ui->textEditDevice->toPlainText(); //整个内容作为字符串
    QByteArray strBytes=str.toUtf8(); //转换为字节数组
    aFile.write(strBytes,strBytes.length()); //写入文件
    aFile.close();
    ui->tabWidget->setCurrentIndex(0);
    return true;
}

```

自定义函数 `saveTextByIODevice()` 实现文件保存功能，为了保存文件，用 `open()` 打开文件时，使用的模式是 `QIODevice::WriteOnly | QIODevice::Text`。使用 `WriteOnly` 隐含着 `Truncate`，即删除文件原有内容。

首先将 `QPlainTextEdit` 组件 `textEditDevice` 的文本导出为一个字符串，将 `QString` 类的 `toUtf8()` 函数转换为 UTF8 编码的字节数组 `strBytes`，然后调用 `QFile::write()` 函数将字节数组内容写入文件。

### 7.1.3 QFile 和 QTextStream 结合读写文本文件

`QTextStream` 与 IO 读写设备结合，为数据读写提供了一些方便的方法的类，`QTextStream` 可以与 `QFile`、`QTemporaryFile`、`QBuffer`、`QTcpSocket` 和 `QUdpSocket` 等 IO 设备类结合使用。

在本例中，将 `QFile` 和 `QTextStream` 结合，读取文本文件的自定义函数 `openTextByStream()` 的代码如下：

```

bool MainWindow::openTextByStream(const QString &aFileName)
{ //用 QTextStream 打开文本文件
    QFile aFile(aFileName);
    if (!aFile.exists()) //文件不存在
        return false;
    if (!aFile.open(QIODevice::ReadOnly | QIODevice::Text))
        return false;
    QTextStream aStream(&aFile); //用文本流读取文件
    aStream.setAutoDetectUnicode(true); //自动检测 Unicode, 才能显示汉字
    ui->textEditStream->setPlainText(aStream.readAll());
    aFile.close(); //关闭文件
    ui->tabWidget->setCurrentIndex(1);
    return true;
}

```

在创建 `QTextStream` 实例时传递一个 `QFile` 对象，这样，`QFile` 对象和 `QTextStream` 对象就结合在一起了，利用 `QTextStream` 可读写文件。如果文本文件里有汉字，需要设定为自动识别 Unicode 码，即调用 `setAutoDetectUnicode(true)` 函数。

在这段代码里，使用 `QTextStream::readAll()` 函数一次读出文件全部文本内容。但是 `QTextStream` 还提供了一些其他方便使用的接口函数，如使用 `QTextStream` 可以方便地实现逐行读取文本文件内容。对 `openTextByStream()` 函数的内容稍作修改，使其以逐行的方式读取文件内容，这种方式适用于需要逐行解析字符串的内容的应用，如 5.4 节的实例。

```
bool MainWindow::openTextByStream(const QString &aFileName)
{ //用 QTextStream 打开文本文件
    QFile aFile(aFileName);
    if (!aFile.exists()) //文件不存在
        return false;
    if (!aFile.open(QIODevice::ReadOnly | QIODevice::Text))
        return false;
    QTextStream aStream(&aFile); //用文本流读取文件
    aStream.setAutoDetectUnicode(true); //自动检测 Unicode
    ui->textEditStream->clear();
    while (!aStream.atEnd())
    {
        str=aStream.readLine(); //读取文件的一行文本
        ui->textEditStream->appendPlainText(str);
    }
    aFile.close(); //关闭文件
    ui->tabWidget->setCurrentIndex(1);
    return true;
}
```

`QTextStream::readLine()` 函数通过自动识别换行符来读取一行字符串。

`saveTextByStream()` 使用 `QTextStream` 保存文件的自定义函数，代码如下：

```
bool MainWindow::saveTextByStream(const QString &aFileName)
{ //用 QTextStream 保存文本文件
    QFile aFile(aFileName);
    if (!aFile.open(QIODevice::WriteOnly | QIODevice::Text))
        return false;
    QTextStream aStream(&aFile); //用文本流读取文件
    aStream.setAutoDetectUnicode(true); //自动检测 Unicode
    QString str=ui->textEditStream->toPlainText();
    aStream<<str; //写入文本流
    aFile.close(); //关闭文件
    return true;
}
```

因为在写入文件时，直接使用了流的写入操作，所以，使用 `QTextStream` 进行文件读写是比较方便的。

## 7.1.4 解决中文乱码的问题

在使用 `QTextStream` 读写有中文内容的文本文件时，为了能正确识别 Unicode 码，需要调用



setAutoDetectUnicode(true), 设置 QTextStream 可以自动识别 Unicode 码, 如果不做此设置, 读取文件的中文将是乱码, 无法正常显示。

为解决 Unicode 的识别问题, 可以在应用程序中做全局的设置, 使得应用程序支持 Unicode。方法是在 main() 函数中使用 QTextCodec 类进行编码设置。例如, 本实例的 main() 函数如下:

```
int main(int argc, char *argv[])
{
    //解决汉字乱码问题
    QTextCodec *codec = QTextCodec::codecForName("UTF-8");
    QTextCodec::setCodecForLocale(codec);
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

使用 UTF-8 的编码解码器在 main() 函数的前面增加了两行, 并设置为应用程序使用的编码解码器, 这样在应用程序内就有了对 Unicode 码的支持。在前面的 openTextByStream() 函数中, 即使不用 setAutoDetectUnicode(true) 也可以正常显示汉字了。

## 7.2 二进制文件读写

### 7.2.1 实例功能概述

除了文本文件之外, 其他需要按照一定的格式定义读写的文件都称为二进制文件。每种格式的二进制文件都有自己的格式定义, 写入数据时按照一定的顺序写入, 读出时也按照相应的顺序读出。例如地球物理中常用的 SEG-Y 格式文件, 必须按照其标准格式要求写入数据才符合这种文件的格式规范, 读取数据时也需要按照格式定义来读出。

Qt 使用 QFile 和 QDataStream 进行二进制数据文件的读写。QFile 负责文件的 IO 设备接口, 即与文件的物理交互, QDataStream 以数据流的方式读取文件或写入文件内容。

本节以实例 samp7\_2 演示二进制文件的读写, 图 7-2 是程序运行的界面。



图 7-2 实例 samp7\_2 的二进制文件读写功能

实例以表格形式编辑一个数据表，采用 Model/View 结构，编辑后的数据保存为二进制文件，这与第 5.4 节的实例用纯文本文件存储数据不同。

根据 QDataStream 保存文件时使用的数据编码的方式不同，可以保存为两种文件。

(1) 用 Qt 预定义编码保存各种类型数据的文件，定义文件后缀为“.stm”。Qt 预定义编码是指在写入某个类型数据，如整形数、字符串等到文件流时，使用 Qt 预定义的编码。可以将这种 Qt 预定义数据格式编码类比于 HTML 的标记符，Qt 写入某种类型数据时用了 Qt 预定义的标记符，读出数据时，根据标记符读出数据。使用 Qt 预定义编码保存的流文件，某些字节是 QDataStream 自己写入的，我们并不完全知道文件内每个字节的意义，但是用 QDataStream 可以读出相应的数据。

(2) 标准编码数据文件，定义文件后缀为“.dat”。在将数据写到文件时，完全使用数据的二进制原始内容，每个字节都有具体的定义，在读出数据时，只需根据每个字节的定义读出数据即可。

实例 samp7\_2 具有如下功能：

- 可以在表格内编辑数据，同样的表格数据内容可以保存为两种格式的文件，Qt 预定义编码文件（stm 文件）和标准编码文件（dat 文件）；
- 界面上的表格数据可以修改，可以添加行、插入行、删除行；
- 可以读取 stm 文件或 dat 文件，虽然文件格式不一样，但对相同的界面数据表存储的文件的实质内容是一样的。

实例 samp7\_2 的主窗口使用了 Model/View 结构、标准项数据模型 QStandardItemModel 和选择模型 QItemSelectionModel，界面上使用了 QTableView 组件，还有代理组件。这些涉及 Model/View 的设计可参考第 5.4 节和 5.5 节，这些设计在前述章节里已经介绍过，不是本节的重点，不再详述。

为便于理解后面的程序，这里给出主窗口 MainWindow 类中自定义的一些变量和函数，具体如下（忽略了自动生成的一些定义）：

```
class MainWindow : public QMainWindow
{
private:
    QLabel *LabCellPos;                //当前单元格行列号
    QLabel *LabCellText;              //当前单元格内容
    QWIntSpinDelegate intSpinDelegate; //整形数，代理组件
    QWFloatSpinDelegate floatSpinDelegate; //浮点数，代理组件
    QWComboBoxDelegate comboBoxDelegate; //列表选择，代理组件
    QStandardItemModel *theModel;      //数据模型
    QItemSelectionModel *theSelection; //选择模型
    void resetTable(int aRowCount);    //表格复位，设定行数
    bool saveDataAsStream(QString& aFileName); //保存为 stm 文件
    bool openDataAsStream(QString& aFileName); //打开 stm 文件
    bool saveBinaryFile(QString& aFileName); //保存为 dat 文件
    bool openBinaryFile(QString& aFileName); //打开 dat 文件
};
```

## 7.2.2 Qt 预定义编码文件的读写

### 1. 保存为 stm 文件

先看文件保存功能，因为从文件保存功能的代码可以看出文件内数据的存储顺序。在图 7-2

的窗口上编辑表格的数据后，单击工具栏上的“保存 stm 文件”，可以使用 Qt 预定义编码方式保存文件。此按钮的响应代码如下：

```
void MainWindow::on_actSave_triggered()
{ //以 Qt 预定义编码保存文件
    QString curPath=QDir::currentPath();
    QString aFileName=QFileDialog::getSaveFileName(this,"选择保存文件",
        curPath,"Qt 预定义编码数据文件 (*.stm)");
    if (aFileName.isEmpty())
        return;
    if (saveDataAsStream(aFileName))
        QMessageBox::information(this,"提示消息","文件已经成功保存!");
}

bool MainWindow::saveDataAsStream(QString &aFileName)
{ //将模型数据保存为 Qt 预定义编码的数据文件
    QFile aFile(aFileName);
    if (!(aFile.open(QIODevice::WriteOnly | QIODevice::Truncate)))
        return false;
    QDataStream aStream(&aFile);
    aStream.setVersion(QDataStream::Qt_5_9); //流版本号，写入和读取版本要兼容
    qint16 rowCount=theModel->rowCount();
    qint16 colCount=theModel->columnCount();
    aStream<<rowCount; //写入文件流，行数
    aStream<<colCount; //写入文件流，列数
    //获取表头文字
    for (int i=0;i<theModel->columnCount();i++)
    {
        QString str=theModel->horizontalHeaderItem(i)->text(); //获取表头文字
        aStream<<str; //字符串写入文件流
    }
    //获取数据区的数据
    for (int i=0;i<theModel->rowCount();i++)
    {
        QStandardItem* aItem=theModel->item(i,0); //测深
        qint16 ceShen=aItem->data(Qt::DisplayRole).toInt();
        aStream<<ceShen; //写入文件流，qint16
        aItem=theModel->item(i,1); //垂深
        qreal chuiShen=aItem->data(Qt::DisplayRole).toFloat();
        aStream<<chuiShen; //写入文件流，qreal
        aItem=theModel->item(i,2); //方位
        qreal fangWei=aItem->data(Qt::DisplayRole).toFloat();
        aStream<<fangWei; //写入文件流，qreal
        aItem=theModel->item(i,3); //位移
        qreal weiYi=aItem->data(Qt::DisplayRole).toFloat();
        aStream<<weiYi; //写入文件流，qreal
        aItem=theModel->item(i,4); //固井质量
        QString zhiLiang=aItem->data(Qt::DisplayRole).toString();
        aStream<<zhiLiang; //写入文件流，字符串
        aItem=theModel->item(i,5); //测井取样
        bool quYang=(aItem->checkState()==Qt::Checked);
        aStream<<quYang; //写入文件流，bool
    }
    aFile.close();
    return true;
}
```



自定义函数 `saveDataAsStream()` 将表格的数据模型 `theModel` 的数据保存为一个 `stm` 文件。代码首先是创建 `QFile` 对象 `aFile` 打开文件，然后创建 `QDataStream` 对象 `aStream` 与 `QFile` 对象关联。

在开始写数据流之前，为 `QDataStream` 对象 `aStream` 设置版本号，即调用 `setVersion()` 函数，并传递一个 `QDataStream::Version` 枚举类型的值。

```
aStream.setVersion(QDataStream::Qt_5_9);
```

这表示 `aStream` 将以 `QDataStream::Qt_5_9` 版本的预定义类型写文件流。

**注意** 以 Qt 的预定义类型编码保存的文件需要指定流版本号，因为每个版本的 Qt 对数据类型的编码可能有差别，需要保证写文件和读文件的流版本是兼容的。

接下来，就是按照需要保存数据的顺序写入文件流。例如在文件开始，先写入行数和列数两个 `qint16` 的整数。因为行数和列数关系到后面的数据是如何组织的，因此在读取文件数据时，首先读取这两个整数，然后根据数据存储方式的约定，就知道后续数据该如何读取了。向文件写入数据时，直接用流的输入操作，如：

```
aStream<<rowCount; //写入文件流，行数
aStream<<colCount; //写入文件流，列数
```

在读取各列的表头字符串之后，将其写入数据流。然后逐行扫描表格的数据模型，将每一行的列数据写入数据流。

数据流写入数据时都使用运算符“<<”，不论写的是 `qint16`、`qreal`，还是字符串。除了可以写入基本的数据类型外，`QDataStream` 流操作还可以写入很多其他类型的数据，如 `QBrush`、`QColor`、`QImage`、`QIcon` 等，这些称为可序列化的数据类型（Serializing Qt Data Types）。

`QDataStream` 以流操作写入这些数据时，我们并不知道文件里每个字节是如何存储的，但是知道数据写入的顺序，以及每次写入数据的类型。在文件数据读出时，只需按照顺序和类型对应读出即可。

## 2. stm 文件格式

根据 `saveDataAsStream()` 函数的代码，可知 Qt 预定义编码保存的 `stm` 文件的格式，如表 7-1 所示。

表 7-1 以 Qt 预定义编码保存的 `stm` 文件的格式定义

顺序号	数据	类型	备注
1	<code>rowCount</code>	<code>qint16</code>	行数
2	<code>colCount</code>	<code>qint16</code>	列数
3	"Depth"	<code>QString</code>	表头标题 1
4	"Measured Depth"	<code>QString</code>	表头标题 2
5	"Direction"	<code>QString</code>	表头标题 3
6	"Offset"	<code>QString</code>	表头标题 4
7	"Quality"	<code>QString</code>	表头标题 5
8	"Sampled"	<code>QString</code>	表头标题 6
9	第 1 行各列数据	<code>qint16</code>	测深
10		<code>qreal</code>	垂深
11		<code>qreal</code>	方位



续表

顺序号	数据	类型	备注
12		qreal	位移
13		QString	固井质量
14		bool	是否测井取样
15	第 2 行各列数据		
...			

从表 7-1 中可以知道 stm 文件的数据存储顺序和类型，但是并不知道 qint16 类型的数据存储为几个字节以及 QString 类型的数据是如何定义长度和字符内容的，其实也不需要知道这些具体的存储方式，在从文件读出时，只需按照表 7-1 的顺序和类型读出数据即可。

3. 读取 stm 文件

下面是工具栏按钮“打开 stm 文件”的响应代码及相关函数代码，选择需要打开的 stm 文件后，主要是调用自定义函数 openDataAsStream()将其打开。

```
void MainWindow::on_actOpen_triggered()
{ //打开 stm 文件
    QString curPath=QDir::currentPath();
    QString aFileName=QFileDialog::getOpenFileName(this,"打开一个文件",
        curPath,"Qt 预定义编码数据文件 (*.stm)");
    if (aFileName.isEmpty())
        return;
    if (openDataAsStream(aFileName))
        QMessageBox::information(this,"提示消息","文件已经打开!");
}

bool MainWindow::openDataAsStream(QString &aFileName)
{ //从 stm 文件读入数据
    QFile aFile(aFileName);
    if (!(aFile.open(QIODevice::ReadOnly)))
        return false;
    QDataStream aStream(&aFile);
    aStream.setVersion(QDataStream::Qt_5_9); //设置流文件版本号
    qint16 rowCount,colCount;
    aStream>>rowCount; //读取行数
    aStream>>colCount; //读取列数
    this->resetTable(rowCount); //表格复位,设定行数
    //获取表头文字,但并不使用
    QString str;
    for (int i=0;i<colCount;i++)
        aStream>>str; //读取表头字符串
    //读取数据区数据
    qint16 ceShen;
    qreal chuiShen, fangWei, weiYi;
    QString zhiLiang;
    bool quYang;
    QStandardItem *aItem;
    QModelIndex index;
    for (int i=0;i<rowCount;i++)
    {
        aStream>>ceShen; //读取测深, qint16
        index=theModel->index(i,0);
```

```

        aItem=theModel->itemFromIndex(index);
        aItem->setData(ceShen,Qt::DisplayRole);
        aStream>>chuiShen;//垂深,qreal
        index=theModel->index(i,1);
        aItem=theModel->itemFromIndex(index);
        aItem->setData(chuiShen,Qt::DisplayRole);
        aStream>>fangWei;//方位,qreal
        index=theModel->index(i,2);
        aItem=theModel->itemFromIndex(index);
        aItem->setData(fangWei,Qt::DisplayRole);
        aStream>>weiYi;//位移,qreal
        index=theModel->index(i,3);
        aItem=theModel->itemFromIndex(index);
        aItem->setData(weiYi,Qt::DisplayRole);
        aStream>>zhiLiang;//固井质量,QString
        index=theModel->index(i,4);
        aItem=theModel->itemFromIndex(index);
        aItem->setData(zhiLiang,Qt::DisplayRole);
        aStream>>quYang;//测井取样
        index=theModel->index(i,5);
        aItem=theModel->itemFromIndex(index);
        if (quYang)
            aItem->setCheckState(Qt::Checked);
        else
            aItem->setCheckState(Qt::Unchecked);
    }
    aFile.close();
    return true;
}

void MainWindow::resetTable(int aRowCount)
{ //表格复位
    theModel->removeRows(0,theModel->rowCount());
    theModel->setRowCount(aRowCount);
    QString str=theModel->headerData(theModel->columnCount()-1,
        Qt::Horizontal,Qt::DisplayRole).toString();
    for (int i=0;i<theModel->rowCount();i++)
    { //设置最后一列
        QModelIndex index=theModel->index(i,FixedColumnCount-1);
        QStandardItem* aItem=theModel->itemFromIndex(index);
        aItem->setCheckable(true);
        aItem->setData(str,Qt::DisplayRole);
        aItem->setEditable(false);
    }
}

```

读取 stm 文件的数据之前也必须设置 QDataStream 的流版本号，应该等于或高于数据保存时的流版本号。

然后就是按照表 7-1 所示的写入数据时的顺序和类型，相应地读出每个数据。文件里最早的两个数据是表格的行数和列数，读出这两个数据，就能知道数据的行数和列数，并调用自定义函数 resetTable() 给数据模型复位，并设置其行数。

然后将保存的每行数据读入到数据模型的每个项中，这样窗口上的 QTableView 组件就可以显示数据了。

使用 QDataStream 的流操作方式读写文件的特点如下。

- 读写操作都比较方便，支持读写各种数据类型，包括 Qt 的一些类，还可以为流数据读写扩展自定义的数据类型。读写某种类型的数据时，只要是流支持即可，而在文件内部是如何存储的，用户无需关心，由 Qt 预定义。
- 写文件和读文件时必须保证使用的流版本兼容，即流的版本号相同，或读取文件的流版本号高于写文件时的流版本号。这是因为在不同的流版本中，流支持的数据类型的读写方式可能有所改变，必须保证读写版本的兼容。
- 用这种方式保存文件时，写入数据采用 Qt 预定义的编码，即写入文件的二进制编码是由 Qt 预定义的，写多少个字节、字节是什么样的顺序，用户是不知道的。如果是由 QDataStream 读取数据，只需按类型读出即可。但是，如果由这种方法创建的文件是用于交换的，需要用其他的编程语言（如 Matlab）来读取文件内容，则存在问题了。因为其他语言并没有与 Qt 的流写入完全一致的流读出功能，例如，其他语言并不知道 Qt 保存的 QString 或 QFont 的内容是如何组织的。

### 7.2.3 标准编码文件的读写

#### 1. 保存为 dat 文件

前面是采用 Qt 预定义编码读写 stm 文件，这种方法使用简单，但是文件的格式不完全透明，不能创建用于交换的通用格式文件。

创建通用格式文件（即文件格式完全透明，每个字节都有具体的定义，如 SEG-Y 文件）的方法是以标准编码方式创建文件，使文件的每个字节都有具体的定义。用户在读取这种文件时，按照文件格式定义读取出每个字节数据并做解析即可，不管使用什么编程语言都可以编写读写文件的程序。

主窗口工具栏上的“保存 dat 文件”按钮将表格中的数据保存为标准编码的文件，文件后缀是“.dat”。保存 dat 文件的代码是：

```
void MainWindow::on_actSaveBin_triggered()
{ //保存 dat 文件
    QString curPath=QDir::currentPath();
    QString aFileName=QFileDialog::getSaveFileName(this,"选择保存文件",
                                                    curPath, "标准编码数据文件 (*.dat)");
    if (aFileName.isEmpty())
        return;
    if (saveBinaryFile(aFileName))
        QMessageBox::information(this,"提示消息","文件已经成功保存!");
}

bool MainWindow::saveBinaryFile(QString &aFileName)
{ //保存为 dat 文件
    QFile aFile(aFileName);
    if (!(aFile.open(QIODevice::WriteOnly)))
        return false;
    QDataStream aStream(&aFile);
    aStream.setByteOrder(QDataStream::LittleEndian); //小端字节序
    qint16 rowCount=theModel->rowCount();
```

```

qint16 colCount=theModel->columnCount();
aStream.writeRawData((char *)&rowCount,sizeof(qint16)); //写入文件流
aStream.writeRawData((char *)&colCount,sizeof(qint16)); //写入文件流
//获取表头文字
QByteArray btArray;
QStandardItem *aItem;
for (int i=0;i<theModel->columnCount();i++)
{
    aItem=theModel->horizontalHeaderItem(i); //获取表头 item
    QString str=aItem->text(); //获取表头文字
    btArray=str.toUtf8(); //转换为字符数组
    aStream.writeBytes(btArray,btArray.length()); //写入字符串数据
}
//获取表格数据区
qint8 yes=1,no=0; //分别代表逻辑值 true 和 false
for (int i=0;i<theModel->rowCount();i++)
{
    aItem=theModel->item(i,0); //测深
    qint16 ceShen=aItem->data(Qt::DisplayRole).toInt();
    aStream.writeRawData((char *)&ceShen,sizeof(qint16));
    aItem=theModel->item(i,1); //垂深
    qreal chuiShen=aItem->data(Qt::DisplayRole).toFloat();
    aStream.writeRawData((char *)&chuiShen,sizeof(qreal));
    aItem=theModel->item(i,2); //方位
    qreal fangWei=aItem->data(Qt::DisplayRole).toFloat();
    aStream.writeRawData((char *)&fangWei,sizeof(qreal));
    aItem=theModel->item(i,3); //位移
    qreal weiYi=aItem->data(Qt::DisplayRole).toFloat();
    aStream.writeRawData((char *)&weiYi,sizeof(qreal));
    aItem=theModel->item(i,4); //固井质量
    QString zhiLiang=aItem->data(Qt::DisplayRole).toString();
    btArray=zhiLiang.toUtf8();
    aStream.writeBytes(btArray,btArray.length()); //写字符串数据
    aItem=theModel->item(i,5); //测井取样
    bool quYang=(aItem->checkState()==Qt::Checked);
    if (quYang)
        aStream.writeRawData((char *)&yes,sizeof(qint8));
    else
        aStream.writeRawData((char *)&no,sizeof(qint8));
}
aFile.close();
return true;
}

```

## • 字节序

在保存为标准编码的二进制文件时，无须指定 QDataStream 的版本，因为不会用到 Qt 的类型预定义编码，文件的每个字节的意义都是用户自己定义的。但是如有必要，需要为文件指定字节顺序，如：

```
aStream.setByteOrder(QDataStream::LittleEndian);
```

字节顺序分为大端字节序和小端字节序，小端字节序指低字节数据存放在内存低地址处，高字节数据存放在内存高地址处；大端字节序则相反。

基于 X86 平台的计算机是小端字节序的，所以 Windows 系统是小端字节序，而有的嵌入式平



台或工作站平台则是大端字节序的。读取一个文件时，首先需要知道它是以什么字节序存储的，这样才可以正确的读出。

setByteOrder()函数的参数是 QDataStream::ByteOrder 枚举类型常量，QDataStream::BigEndian 是大端字节序，QDataStream::LittleEndian 是小端字节序。

• writeRawData()函数

QDataStream 采用函数 writeRawData()将数据写入数据流，在保存 qint8、qint16、qreal 等类型的数据时都使用这个函数，其函数原型是：

```
int QDataStream::writeRawData(const char *s, int len)
```

其中参数 s 是一个指向字节型数据的指针，len 是字节数据的长度。调用 writeRawData()函数将会向文件流连续写入 len 个字节的数据，这些字节数据保存在指针 s 指向的起始地址里。例如，将 qint16 类型变量 rowCount 写入文件的语句是：

```
qint16 rowCount=theModel->rowCount();
aStream.writeRawData((char *)&rowCount,sizeof(qint16));
```

• writeBytes()函数

在将字符串数据写入文件时，使用的是 writeBytes()函数，而不是 writeRawData()。下面是 writeBytes()函数的原型定义：

```
QDataStream &QDataStream::writeBytes(const char *s, uint len)
```

其中参数 s 是一个指向字节型数据的指针，len 是字节数据的长度。writeBytes()在写入数据时，会先将 len 作为一个 quint32 类型写入数据流，然后再写入 len 个从指针 s 获取的数据。

writeBytes()适合于写入字符串数据，因为在写入字符串之前要先写入字符串的长度，这样在读取文件时，就能知道字符串的长度，以便正确读出字符串。

例如，下面的代码将字符串“Depth”写入文件流：

```
QString str="Depth";
QByteArray btArray=str.toUtf8();
aStream.writeBytes(btArray,btArray.length());
```

文件中实际保存的内容见表 7-2。前 4 个字节是 quint32 类型的整数，表示保存数据的字节个数，这里是 5，表示后续有 5 个字节数据。从第 5 字节开始，是保存的字符串“Depth”的每个字符的 ASCII 码。

表 7-2 writeBytes()保存内容示例

字节序号	1	2	3	4	5	6	7	8	9
字节数据 (Hex)	05	00	00	00	44	65	70	74	68
内容	后续数据长度，表示有 5 字节				D	e	p	t	h

由于写入文件的字符串的长度一般是不固定的，因此如果以 writeRawData()函数写入文件，只会写入字符串的内容，而没有表示字符串的长度。在文件读出时，如果不已知字符串长度，则难以正确读出字符串内容。而 writeBytes()函数首先写入了字符串的长度，在读取文件时，先从前四个字节读出字符串长度，知道数据有多少个字节就可以正确读出了。

QDataStream 提供了与 writeBytes()对应的函数 readBytes(), 它可以自动读取长度和内容, 适用于字符串数据的读取。

2. dat 文件格式

用 saveBinaryFile() 函数保存数据为标准编码二进制文件, 文件后缀为 “.dat”。根据 saveBinaryFile()函数的内容, dat 文件的格式见表 7-3。

表 7-3 标准编码保存的 dat 文件的格式定义

顺序号	数据	类型	字节数	备注
1	rowCount	qint16	2	行数
2	colCount	qint16	2	列数
3	"Depth"	QString	4+5	表头标题 1
4	"Measured Depth"	QString	4+14	表头标题 2
5	"Direction"	QString	4+9	表头标题 3
6	"Offset"	QString	4+6	表头标题 4
7	"Quality"	QString	4+7	表头标题 5
8	"Sampled"	QString	4+7	表头标题 6
9	第 1 行各列数据	qint16	2	测深
10		qreal	8	垂深
11		qreal	8	方位
12		qreal	8	位移
13		QString	4+字符串字节数	固井质量字符串
14		qint8	1	是否测井取样
15	第 2 行各列数据			
...				

在表 7-3 中, 可以看到文件内的每个字节都是有具体定义的, 这样, 无论用什么语言编写一个文件读取的程序, 只要按照这个格式来读取, 都可以正确读出文件内容。

dat 文件的数据是否是按照表 7-3 所示的顺序存储的呢? 可以创建一个简单的数据表格, 保存为 dat 后缀的文件, 然后用显示文件二进制内容的软件来查看, 如 UltraEdit 或 WinHex, 这些软件在分析文件格式, 编写文件读写程序时特别有用。

3. 读取 dat 文件

对于保存的 dat 文件, 主窗口工具栏上的“打开 dat 文件”按钮可以打开保存的 dat 文件, 下面是打开 dat 文件的函数 openBinaryFile()的代码。

```
bool MainWindow::openBinaryFile(QString &aFileName)
{
    //打开 dat 文件
    QFile aFile(aFileName);
    if (!(aFile.open(QIODevice::ReadOnly)))
        return false;
    QDataStream aStream(&aFile);
    aStream.setByteOrder(QDataStream::LittleEndian);

    qint16 rowCount,colCount;
    aStream.readRawData((char *)&rowCount, sizeof(qint16));
    aStream.readRawData((char *)&colCount, sizeof(qint16));
    this->resetTable(rowCount);
    //读取表头文字, 但是并不利用
    char *buf;
```

```

uint strLen;
for (int i=0;i<colCount;i++)
{
    aStream.readBytes(buf, strLen); //同时读取字符串长度和字符串内容
    QString str=QString::fromLocal8Bit(buf, strLen);
}
//读取数据区数据
QStandardItem *aItem;
qint16 ceShen;
qreal chuiShen, fangWei, weiYi;
QString zhiLiang;
qint8 quYang; //1==true, 0==false
QModelIndex index;
for (int i=0;i<rowCount;i++)
{
    aStream.readRawData((char *)&ceShen, sizeof(qint16)); //测深
    index=theModel->index(i, 0);
    aItem=theModel->itemFromIndex(index);
    aItem->setData(ceShen, Qt::DisplayRole);
    aStream.readRawData((char *)&chuiShen, sizeof(qreal)); //垂深
    index=theModel->index(i, 1);
    aItem=theModel->itemFromIndex(index);
    aItem->setData(chuiShen, Qt::DisplayRole);
    aStream.readRawData((char *)&fangWei, sizeof(qreal)); //方位
    index=theModel->index(i, 2);
    aItem=theModel->itemFromIndex(index);
    aItem->setData(fangWei, Qt::DisplayRole);
    aStream.readRawData((char *)&weiYi, sizeof(qreal)); //位移
    index=theModel->index(i, 3);
    aItem=theModel->itemFromIndex(index);
    aItem->setData(weiYi, Qt::DisplayRole);
    aStream.readBytes(buf, strLen); //固井质量
    zhiLiang=QString::fromLocal8Bit(buf, strLen);
    index=theModel->index(i, 4);
    aItem=theModel->itemFromIndex(index);
    aItem->setData(zhiLiang, Qt::DisplayRole);
    aStream.readRawData((char *)&quYang, sizeof(qint8)); //测井取样
    index=theModel->index(i, 5);
    aItem=theModel->itemFromIndex(index);
    if (quYang==1)
        aItem->setCheckState(Qt::Checked);
    else
        aItem->setCheckState(Qt::Unchecked);
}
aFile.close();
return true;
}

```

### • 字节序

在流创建后, 需要用 `setByteOrder()` 函数指定字节序, 并且与写入文件时用的字节序一致。

### • readRawData() 函数

在读取基本类型数据时, 使用 `QDataStream` 的 `readRawData()` 函数, 该函数原型为:

```
int QDataStream::readRawData(char *s, int len)
```

它会读取 `len` 个字节的数据，并且保存到指针 `s` 指向的存储区。例如：

```
qint16 rowCount;
aStream.readRawData((char *)&rowCount, sizeof(qint16));
```

- `readBytes()` 函数

读取字符串时使用 `readBytes()` 函数，它是与 `writeBytes()` 功能对应的函数，其函数原型为：

```
QDataStream &QDataStream::readBytes(char *&s, uint &len)
```

对应表格 7-2，使用 `readBytes()` 函数时，会先自动读取前 4 个字节数据作为 `quint32` 的数据，并赋值给 `len` 参数，因为 `len` 是以引用方式传递的参数，所以，`len` 返回读取的数据的字节数。然后根据 `len` 的大小读取相应字节的数据，存储到指针 `s` 指向的存储区。

## 7.3 文件目录操作

### 7.3.1 文件目录操作相关的类

Qt 为文件和目录操作提供了一些类，利用这些类可以方便地实现一些操作。Qt 提供的与文件和目录操作相关的类包括以下几个。

- `QCoreApplication`：用于提取应用程序路径、程序名等文件信息。
- `QFile`：除了打开文件操作外，`QFile` 还有复制文件、删除文件等功能。
- `QFileInfo`：用于提取文件的信息，包括路径、文件名、后缀等。
- `QDir`：用于提取目录或文件信息，获取一个目录下的文件或目录列表，创建或删除目录和文件，文件重命名等操作。
- `QTemporaryDir` 和 `QTemporaryFile`：用于创建临时目录和临时文件。
- `QFileSystemWatcher`：文件和目录监听类，监听目录下文件的添加、删除等变化，监听文件修改变化。

这些类基本涵盖了文件操作需要的主要功能，有些功能还在某些类里重复出现，例如 `QFile` 和 `QDir` 都具有删除文件、判断文件是否存在的功能。

### 7.3.2 实例概述

#### 1. 实例功能

实例 `samp7_3` 演示前述各种目录与文件操作类的主要功能，图 7-3 是实例运行时的窗口。窗口左侧是一个 `QToolBox` 组件，分为 6 组，每一组是一个或两个类的功能演示，在每个组里放置一些 `QPushButton` 按钮，每个按钮主要调用类的某个函数，按钮的标题就是使用的函数的名称。

窗口右侧是显示区，可以选择一个目录、一个文件，然后左侧的功能基本上都是对选择的目录或文件进行操作，右下方是一个 `QPlainTextEdit` 组件，用于显示信息。

#### 2. 信号发射者信息的获取

每个按钮一般用函数名称作为标题，例如“`QFileInfo` 类”分组里的按钮“`baseName()`”是要



演示 QFileInfo 的 baseName() 函数。另外，将 Qt 帮助文件里的这个函数的基本描述文字复制作为按钮的 ToolTip 文字，例如 “baseName()” 按钮的 ToolTip 属性是 “Returns the base name of the file without the path”。

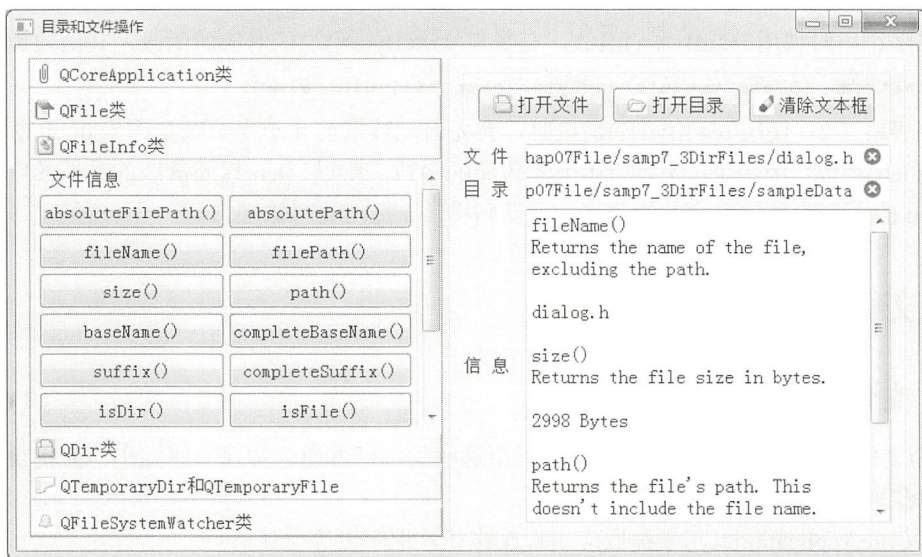


图 7-3 实例 samp7\_3 运行时窗口

在按钮被单击时，先显示按钮的标题和 ToolTip 信息，以便明显地知道按钮演示的功能。例如，“baseName()” 按钮的 clicked() 槽函数代码如下：

```
void Dialog::on_pushButton_30_clicked()
{
    //QFileInfo.basename()
    QPushButton *btn = static_cast<QPushButton*>(sender());
    ui->plainTextEdit->appendPlainText(btn->text());
    ui->plainTextEdit->appendPlainText(btn->tooltip()+"\n");

    QFileInfo fileInfo(ui->editFile->text());
    QString str=fileInfo.baseName();
    ui->plainTextEdit->appendPlainText(str+"\n");
}
```

代码的前 3 行是用于获取按钮对象，并显示按钮的 text 和 tooltip 文字。

这里用到了 QObject::sender() 函数，该函数用于在槽函数里获取发射信号的对象。因为这个函数是按钮的 clicked() 信号的槽函数，所以，sender() 获取的信号发射对象就是这个按钮。然后将此对象转换为 QPushButton 类型变量 btn，就可以访问 btn 的 text() 和 tooltip() 函数了。

这种方法的优点是没有出现对象的 ObjectName，这 3 行代码可以在任何一个按钮的 clicked() 信号槽函数里出现。如果是直接用按钮的 ObjectName，则每个按钮里的代码里需要修改名称。本实例有几十个按钮，工作量很大，也容易遗漏了修改名称。

对上面的代码还可以进一步优化，因为每个按钮的槽函数里都要重复这 3 行代码，将其编写为一个函数，然后在按钮的响应代码里调用。优化后的代码如下：

```
void Dialog::on_pushButton_30_clicked()
{
    QFile::FileInfo fileInfo(sender());
    showBtnInfo(sender());
    QFile::FileInfo fileInfo(ui->editFile->text());
    QString str=fileInfo.baseName();
    ui->plainTextEdit->appendPlainText(str+"\n");
}
void Dialog::showBtnInfo(QObject *btn)
{
    //显示 btn 的信息
    QPushButton *theBtn =static_cast<QPushButton*>(btn);
    ui->plainTextEdit->appendPlainText(theBtn->text());
    ui->plainTextEdit->appendPlainText(theBtn->toolTip()+"\n");
}
```

在主窗口类中定义一个私有函数 showBtnInfo(), 接收 QObject \*btn 对象作为输入参数, 将 QObject 对象转换为 QPushButton 对象, 然后显示按钮的 text 和 tooltip。

在按钮的响应代码里只需用一行语句调用 showBtnInfo()函数, 并将 sender()作为参数传递。  
本实例有几十个按钮, 每个按钮的响应代码的第一行都是 showBtnInfo(sender()), 这样可以大大简化代码。

### 7.3.3 QCoreApplication 类

QCoreApplication 是为无 GUI 应用提供事件循环的类, 是所有应用程序类的基类, 其子类 QGuiApplication 为有 GUI 界面的应用程序提供流控制和主要的设定, QGuiApplication 的子类 QApplication 为基于 QWidget 的应用程序提供支持, 包括界面的初始化等。

创建的 Qt Widget Application 都是基于 QApplication 的, 在 main()函数里可以看到 QApplication 的应用。

QCoreApplication 提供了一些有用的静态函数, 可以获取应用程序的名称、启动路径等信息, 几个函数的名称和功能见表 7-4 (省略了函数参数中的 const 关键字)。

表 7-4 QCoreApplication 的有用函数

函数原型	功能
QString applicationDirPath()	返回应用程序启动路径
QString applicationFilePath()	返回应用程序的带有目录的完整文件名
QString applicationName()	返回应用程序名称, 无路径无后缀
QStringList libraryPaths()	返回动态加载库文件时, 应用程序搜索的目录列表
void setOrganizationName(QString &orgName)	为应用程序设置一个机构名
QString organizationName()	返回应用程序的机构名
void exit()	退出应用程序

### 7.3.4 QFile 类

前两节使用 QFile 类进行文件的操作, 应用了 QFile::open()函数。除了打开文件提供读写操作外, QFile 还有一些静态函数和成员函数用于文件操作。表 7-5 是 QFile 的一些静态函数 (省略了函数参数中的 const 关键字)。

表 7-5 QFile 的一些静态函数

函数原型	功能
bool copy(QString &fileName, QString &newName)	复制文件
bool rename(QString &oldName, QString &newName)	重命名文件
bool remove(QString &fileName)	删除一个文件
bool exists(QString &fileName)	判断文件是否存在
bool setPermissions(QString &fileName, Permissions permissions)	设置文件的权限，权限类型是枚举类型 QFileDevice::Permission
Permissions permissions(QString &fileName)	返回文件的权限

静态函数是无需创建 QFile 类对象实例就可以调用的函数，例如使用静态函数 exists()判断一个文件是否存在的代码如下：

```
void Dialog::on_pushButton_51_clicked()
{
    // QFile::exists() 判断文件是否存在
    showBtnInfo(sender());
    QString sous=ui->editFile->text(); //源文件
    bool the=QFile::exists(sous);
    if(the)
        ui->plainTextEdit->appendPlainText("true \n");
    else
        ui->plainTextEdit->appendPlainText("false \n");
}
```

QFile 还提供了对应的成员函数，见表 7-6（省略了函数参数中的 const 关键字）。

表 7-6 QFile 的一些成员函数

函数原型	功能
void setFileName(QString &name)	设置文件名，文件已打开后不能再调用此函数
bool copy(QString &newName)	当前文件复制为 newName 表示的文件
bool rename(QString &newName)	将当前文件重命名为 newName
bool remove()	删除当前文件
bool exists()	判断当前文件是否存在
bool setPermissions(Permissions permissions)	设置文件权限
Permissions permissions()	返回文件的权限
qint64 size()	返回文件的大小，字节数

创建 QFile 对象时可以在构造函数里指定文件名，也可以用 setFileName()指定文件，但是文件打开后不能再调用 setFileName()函数。指定的文件作为 QFile 对象的当前文件，然后成员函数 copy()、rename()等都是基于当前文件的操作。

7.3.5 QFileInfo 类

QFileInfo 类的接口函数提供文件的各种信息。QFileInfo 对象创建时可以指定一个文件名作为当前文件，也可以用 setFile()函数指定一个文件作为当前文件。

QFileInfo 常见接口函数和功能见表 7-7。除了一个静态函数 exists()之外，其他都是公共接口函数，接口函数的操作都是针对当前文件（省略了函数参数中的 const 关键字）。



表 7-7 QFileInfo 的一些函数

函数原型	功能
void setFile(QString &file)	设置一个文件名, 作为 QFileInfo 操作的文件
QString absoluteFilePath()	返回带有文件名的绝对文件路径
QString absolutePath()	返回绝对路径, 不带文件名
QString fileName()	返回去除路径的文件名
QString filePath()	返回包含路径的文件名
QString path()	返回不含文件名的路径
qint64 size()	返回文件大小, 以字节为单位
QString baseName()	返回文件基名, 第一个 “.” 之前的文件名
QString completeBaseName()	返回文件基名, 最后一个 “.” 之前的文件名
QString suffix()	最后一个 “.” 之后的后缀
QString completeSuffix()	第一个 “.” 之后的后缀
bool isDir()	判断当前对象是否是一个目录或目录的快捷方式
bool isFile()	判断当前对象是否是一个文件或文件的快捷方式
bool isExecutable()	判断当前文件是否是可执行文件
QDateTime created()	返回文件创建时间
QDateTime lastModified()	返回文件最后一次被修改的时间
QDateTime lastRead()	返回文件最后一次被读取的时间
bool exists()	判断文件是否存在
bool exists(QString &file)	静态函数, 判断 file 表示的文件是否存在

QFileInfo 提供的这些函数可以提取文件的信息, 包括目录名、文件基名 (不带后缀)、文件后缀等, 利用这些函数可以实现灵活的文件操作。例如, 下面的代码是利用 QFile::rename() 函数和 QFileInfo 的一些功能实现文件重命名功能的代码, 其中就用到了提取路径、提取文件基名的功能。

```
void Dialog::on_pushButton_50_clicked()
{
    //QFile::rename()
    showBtnInfo(sender());
    QString sous=ui->editFile->text(); //源文件
    QFileInfo fileInfo(sous); //源文件信息
    QString newFile=fileInfo.path()+"/"+fileInfo.baseName()+".XYZ";
    QFile::rename(sous,newFile); //重命名文件,更改后缀名
    ui->plainTextEdit->appendPlainText("源文件: "+sous);
    ui->plainTextEdit->appendPlainText("重命名为: "+newFile+"\n");
}
```

表 7-7 中的函数的使用方法和执行效果不再详细列举和说明, 运行实例 samp7\_3 观察执行结果, 可参考 Qt 帮助文件或 samp7\_3 的源程序看函数使用方法。

### 7.3.6 QDir 类

QDir 是进行目录操作的类, 在创建 QDir 对象时传递一个目录字符串作为当前目录, 然后 QDir 函数就可以针对当前目录或目录下的文件进行操作。表 7-8 是 QDir 的一些静态函数 (省略了函数参数中的 const 关键字)。

表 7-8 QDir 的一些静态函数

函数原型	功能
QString tempPath()	返回临时文件目录名称
QString rootPath()	返回根目录名称



续表

函数原型	功能
QString homePath()	返回主目录名称
QString currentPath()	返回当前目录名称
bool setCurrent(QString &path)	设置 path 表示的目录为当前目录
QFileInfoList drives()	返回系统的根目录列表, 在 Windows 系统上返回的是盘符列表

在使用 QFileDialog 选择打开一个文件或目录时需要传递一个初始目录, 这个时候就可以使用 QDir::currentPath() 获取应用程序当前目录作为初始目录, 前面一些实例程序的代码中已经用到过这个功能。

表 7-9 是 QDir 的一些公共接口函数 (省略了函数参数中的 const 关键字)。

表 7-9 QDir 的一些成员函数

函数原型	功能
QString absoluteFilePath(QString &fileName)	返回当前目录下的一个文件的含绝对路径文件名
QString absolutePath()	返回当前目录的绝对路径
QString canonicalPath()	返回当前目录的标准路径
QString filePath(QString &fileName)	返回目录下一个文件的目录名
QString dirName()	返回最后一级目录的名称
bool exists()	判断当前目录是否存在
QStringList entryList(Filters filters = NoFilter, SortFlags sort = NoSort)	返回目录下的所有文件名、子目录名等
bool mkdir(QString &dirName)	在当前目录下建一个名称为 dirName 的子目录
bool rmdir(QString &dirName)	删除指定的目录 dirName
bool remove(QString &fileName)	删除当前目录下的文件 fileName
bool rename(QString &oldName, QString &newName)	将文件或目录 oldName 更名为 newName
void setPath(QString &path)	设置 QDir 对象的当前目录
bool removeRecursively()	删除当前目录及其下面的所有文件

获取目录下的目录或文件列表的函数 entryList() 需要传递 QDir::Filter 枚举类型的参数以获取不同的结果, QDir::Filter 枚举类型的常用取值如下。

- QDir::AllDirs: 列出所有目录名。
- QDir::Files: 列出所有文件。
- QDir::Drives: 列出所有盘符 (Unix 系统下无效)。
- QDir::NoDotAndDotDot: 不列出特殊的符号, 如 “.” 和 “..”。
- QDir::AllEntries: 列出目录下所有项目。

列出所有子目录的程序如下:

```
void Dialog::on_pushButton_11_clicked()
{
    // 列出子目录
    showBtnInfo(sender());
    QDir dir(ui->editDir->text());
    QStringList strList=dir.entryList(QDir::Dirs | QDir::NoDotAndDotDot);
    ui->plainTextEdit->appendPlainText("所选目录下的所有目录:");
    for(int i=0;i<strList.count();i++)
        ui->plainTextEdit->appendPlainText(strList.at(i));
    ui->plainTextEdit->appendPlainText("\n");
}
```

列出一个目录下所有文件的程序如下：

```
void Dialog::on_pushButton_17_clicked()
{ //列出所有文件
    showBtnInfo(sender());
    QDir dir(ui->editDir->text());
    QStringList strList=dir.entryList(QDir::Files);
    ui->plainTextEdit->appendPlainText("所选目录下的所有文件: ");
    for(int i=0;i<strList.count();i++)
        ui->plainTextEdit->appendPlainText(strList.at(i));
    ui->plainTextEdit->appendPlainText("\n");
}
```

### 7.3.7 QTemporaryDir 和 QTemporaryFile

QTemporaryDir 是用于创建、删除临时目录的类，其主要函数见表 7-10。

表 7-10 QTemporaryDir 的一些成员函数

函数原型	功能
void setAutoRemove(bool b)	设置为是否自动删除
QString path()	返回创建的临时目录名称
bool remove()	删除此临时目录及其下面所有文件

在系统临时目录，即 QDir::tempPath 目录下创建一个临时目录，临时目录名称以 QCoreApplication::applicationName() 为前缀，后加 6 个字符。临时目录可以设置为使用完后自动删除，即临时目录变量删除时，临时目录也删除。

QTemporaryFile 是用于创建临时文件的类，临时文件保存在系统临时目录下。临时文件以 QCoreApplication::applicationName() 作为文件名，以 “XXXXXX” 6 个随机数字作为文件后缀。将 QTemporaryFile::setAutoRemove() 函数设置为是否自动删除临时文件，QTemporaryFile::open() 函数用于打开临时文件，只有打开临时文件，才实际创建了此文件。

### 7.3.8 QFileSystemWatcher 类

QFileSystemWatcher 是对目录和文件进行监听的类。把某些目录或文件添加到 QFileSystemWatcher 对象的监听列表后，当目录下发生文件新建、删除等操作时会发射 directoryChanged() 信号，当监听的文件发生修改、重命名等操作时，会发射 fileChanged() 信号。所以，这个类在进行目录或文件监听时起作用。

QFileSystemWatcher 的主要接口函数见表 7-11（省略了函数参数中的 const 关键字）。

表 7-11 QFileSystemWatcher 的成员函数

函数原型	功能
bool addPath(QString &path)	添加一个监听的目录或文件
QStringList addPaths(QStringList &paths)	添加需要监听的目录或文件列表
QStringList directories()	返回监听的目录列表
QStringList files()	返回监听的文件列表
bool removePath(QString &path)	移除监听的目录或文件
QStringList removePaths(QStringList &paths)	移除监听的目录或文件列表

QFileSystemWatcher 有两个信号，分别是目录变化和文件变化时发射的信号。

```
void QFileSystemWatcher::directoryChanged(const QString &path)
void QFileSystemWatcher::fileChanged(const QString &path)
```

图 7-4 是实例中测试 QFileSystemWatcher 的界面。首先打开一个目录和一个文件，单击“addPath()开始监听”按钮将文件和目录都添加到监听列表，并且将信号与槽函数关联起来。然后在目录下复制某个文件，会发射 directoryChanged()信号，重命名所监听的文件后会发射 fileChanged()信号，如图 7-4 所示的运行结果。

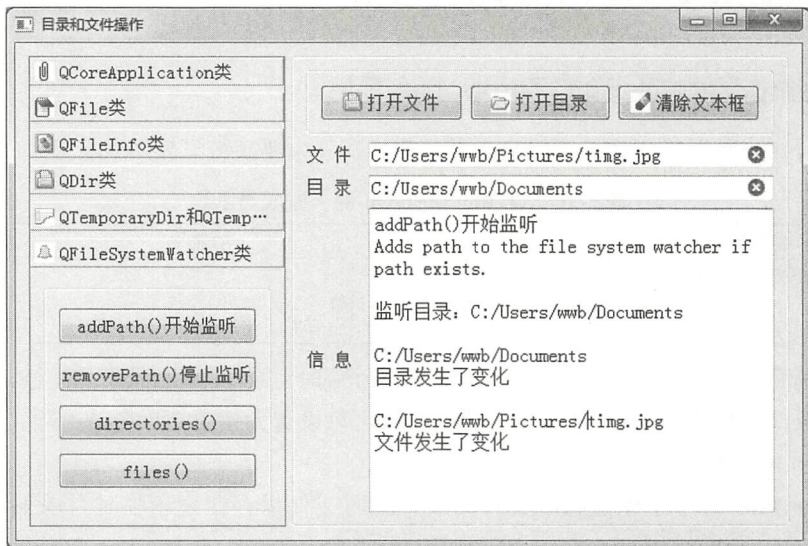


图 7-4 测试 QFileSystemWatcher 的运行界面

为了测试 QFileSystemWatcher 的功能，在主窗口类中定义了 QFileSystemWatcher 类型的变量和两个信号的槽函数，定义如下：

```
private:
    QFileSystemWatcher fileWatcher;
public slots:
    void on_directoryChanged(const QString path);
    void on_fileChanged(const QString path);
```

两个槽函数仅是显示传递的参数，并显示提示文字，其代码如下：

```
void Dialog::on_directoryChanged(const QString path)
{ //directoryChanged() 信号的槽函数
    ui->plainTextEdit->appendPlainText(path);
    ui->plainTextEdit->appendPlainText("目录发生了变化\n");
}
void Dialog::on_fileChanged(const QString path)
{ //fileChanged() 信号的槽函数
    ui->plainTextEdit->appendPlainText(path);
    ui->plainTextEdit->appendPlainText("文件发生了变化\n");
}
```

图 7-4 中 QFileSystemWatcher 分组里“addPath()开始监听”和“removePath()停止监听”两个

按钮的代码如下：

```
void Dialog::on_pushButton_46_clicked()
{
    //开始监听,addPath()
    showBtnInfo(sender());
    ui->plainTextEdit->appendPlainText("监听目录: "+ui->editDir->text()+"\n");
    fileWatcher.addPath(ui->editDir->text()); //添加监听目录
    fileWatcher.addPath(ui->editFile->text()); //添加监听文件
    QObject::connect(&fileWatcher, &QFileSystemWatcher::directoryChanged,
        this, &Dialog::on_directoryChanged); //directoryChanged
    QObject::connect(&fileWatcher, &QFileSystemWatcher::fileChanged,
        this, &Dialog::on_fileChanged); //fileChanged
}

void Dialog::on_pushButton_47_clicked()
{
    //停止监听,removePath()
    showBtnInfo(sender());
    ui->plainTextEdit->appendPlainText("停止监听: "+ui->editDir->text()+"\n");
    fileWatcher.removePath(ui->editDir->text());
    fileWatcher.removePath(ui->editFile->text());
    QObject::disconnect(&fileWatcher, &QFileSystemWatcher::directoryChanged,
        this, &Dialog::on_directoryChanged); //directoryChanged
    QObject::disconnect(&fileWatcher, &QFileSystemWatcher::fileChanged,
        this, &Dialog::on_fileChanged); //fileChanged
}
```

采用 `addPath()` 函数添加目录和文件后，将信号和槽函数关联起来，开始监听。

停止监听时，用 `removePath()` 函数移除监听的目录和文件，并用 `disconnect()` 解除信号与槽的关联。



GUI 用户界面的优势是通过可视化的界面元素为用户提供便利的操作，界面上的按钮、编辑框等各种界面组件其实都是通过绘图而得到的。Qt 的二维绘图基本功能是使用 QPainter 在绘图设备上绘图，绘图设备包括 QWidget、QPixmap 等，通过绘制一些基本的点、线、圆等基本形状组成自己需要的图形，得到的图形是不可交互操作的图形。

Qt 还提供了 Graphics View 架构，使用 QGraphicsView、QGraphicsScene 和各种 QGraphicsItem 类绘图，在一个场景中可以绘制大量图件，且每个图件是可选择、可交互的，如同矢量图编辑软件那样可以操作每个图件。Graphics View 架构为用户绘制复杂的组件化图形提供了便利。

本章首先介绍使用 QPainter 绘图的原理，这是 Qt 绘图的基础，再介绍 Graphics View 架构的原理和使用。

## 8.1 QPainter 基本绘图

### 8.1.1 QPainter 绘图系统

#### 1. QPainter 与 QPaintDevice

Qt 的绘图系统使用户可以在屏幕或打印设备上用相同的 API 绘图，绘图系统基于 QPainter、QPaintDevice 和 QPaintEngine 类。QPainter 是用来进行绘图操作的类，QPaintDevice 是一个可以使用 QPainter 进行绘图的抽象的二维界面，QPaintEngine 给 QPainter 提供在不同设备上绘图的接口。QPaintEngine 类由 QPainter 和 QPaintDevice 内部使用，应用程序一般无需和 QPaintEngine 打交道，除非要创建自己的设备类型。

一般的绘图设备包括 QWidget、QPixmap、QImage 等，这些绘图设备为 QPainter 提供一个“画布”。

#### 2. paintEvent 事件和绘图区

QWidget 类及其子类是最常用的绘图设备，从 QWidget 类继承的类都有 paintEvent() 事件，要在设备上绘图，只需重定义此事件并编写响应代码。创建一个 QPainter 对象获取绘图设备的接口，然后就可以在绘图设备的“画布”上绘图了。

在 paintEvent() 事件里绘图的基本程序结构是：

```
void Widget::paintEvent(QPaintEvent *event)
{
```

```

    QPainter painter(this); //创建与绘图设备关联的 QPainter 对象
    ...//painter 在设备的窗口上画图
}

```

首先创建一个属于本绘图设备的 QPainter 对象 painter, 然后使用这个 painter 在绘图设备的窗口上画图。

QWidget 的绘图区就是其窗口内部区域。如图 8-1 所示是在一个 QWidget 窗口上绘制了一个填充矩形(这个实心矩形及其边框是程序绘制的图形, 其他直线和文字是为说明而加的), 整个窗口内部的矩形区就是 QPainter 可以绘图的区域。

QWidget 的内部绘图区的坐标系如图 8-1 所示, 坐标系统的单位是像素。左上角坐标为(0, 0), 向右是 X 轴正方向, 向下是 Y 轴正方向, 绘图区的宽度由 QWidget::width()函数获取, 高度由 QWidget::height()函数获取, 所以, 绘图区右下角的点的坐标是(width(), height())。这个坐标系是 QWidget 绘图区的局部物理坐标, 称为视口 (viewport) 坐标。相应的还有逻辑坐标, 称为窗口 (window) 坐标, 后面再详细介绍。

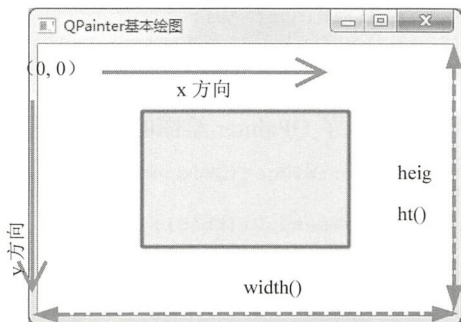


图 8-1 在 QWidget 继承的窗口上绘图

使用 QPainter 在 QWidget 上绘图就是在这样的一个矩形区域里绘图。

### 3. QPainter 绘图的主要属性

用 QPainter 在绘图设备上绘图, 主要是绘制一些基本的图形元素, 包括点、直线、圆形、矩形、曲线、文字等, 控制这些绘图元素特性的主要是 QPainter 的 3 个属性, 分别如下。

- pen 属性: 是一个 QPen 对象, 用于控制线条的颜色、宽度、线型等, 如图 8-1 所示矩形边框的线条的特性就是由 pen 属性决定的。
- brush 属性: 是一个 QBrush 对象, 用于设置一个区域的填充特性, 可以设置填充颜色、填充方式、渐变特性等, 还可以采用图片做材质填充。图 8-1 中的矩形用黄色填充就是由 brush 属性设置决定的。
- font 属性: 是一个 QFont 对象, 用于绘制文字时, 设置文字的字体样式、大小等属性。

使用这 3 个属性基本就控制了绘图的基本特点, 当然还有一些其他的功能结合使用, 比如叠加模式、旋转和缩放等功能。

### 4. 创建实例

为演示 QPainter 绘图的基本功能, 创建一个 Qt Widget Application 项目 samp8\_1, 并选择窗口基类为 QWidget, 自动创建窗体。创建后的项目有一个 Widget 类, 为了简化代码功能, Widget 窗口里不再放置任何其他组件, 只用来绘图。

下面是 Widget 类的完整定义。只是重新定义了 paintEvent()事件, 在此事件里编写绘图的代码。Q\_DECL\_OVERRIDE 宏表示这个函数是对父类虚函数的重载。

```

class Widget : public QWidget
{
    Q_OBJECT
protected:

```

```

    void    paintEvent(QPaintEvent *event) Q_DECL_OVERRIDE;
public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();
private:
    Ui::Widget *ui;
};

```

下面是 Widget 类构造函数和 paintEvent()函数的代码,在界面上绘制如图 8-1 所示的一个填充矩形,演示了 QPainter 绘图的基本过程。

```

Widget::Widget(QWidget *parent) :    QWidget(parent),    ui(new Ui::Widget)
{
    ui->setupUi(this);
    setPalette(QPalette(Qt::white)); //设置窗口为白色背景
    setAutoFillBackground(true);
}
void Widget::paintEvent(QPaintEvent *event)
{
    QPainter    painter    (this); //创建 QPainter 对象
    painter.setRenderHint(QPainter::Antialiasing);
    painter.setRenderHint(QPainter::TextAntialiasing);
    int W=this->width(); //绘图区宽度
    int H=this->height(); //绘图区高度
    QRect    rect(W/4,H/4,W/2,H/2); //中间区域矩形框
    //设置画笔
    QPen    pen;
    pen.setWidth(3); //线宽
    pen.setColor(Qt::red); //划线颜色
    pen.setStyle(Qt::SolidLine); //线的样式,实线、虚线等
    pen.setCapStyle(Qt::FlatCap); //线端点样式
    pen.setJoinStyle(Qt::BevelJoin); //线的连接点样式
    painter.setPen(pen);
    //设置画刷
    QBrush    brush;
    brush.setColor(Qt::yellow); //画刷颜色
    brush.setStyle(Qt::SolidPattern); //画刷填充样式
    painter.setBrush(brush);
    //绘图
    painter.drawRect(rect);
}

```

在 paintEvent()函数中,首先创建与 Widget 关联的 QPainter 对象 painter,这样就可以用这个 painter 在 Widget 上绘图了。

下面的代码获取 Widget 窗口的宽度和高度,并定义了位于中间区域的矩形 rect,这个矩形的大小随 Widget 的大小变化而变化,因为它的大小定义中利用了 Widget 的宽度和高度,而不是固定大小。

```

int W=this->width(); //绘图区宽度
int H=this->height(); //绘图区高度
QRect    rect(W/4,H/4,W/2,H/2); //中间区域矩形框

```

然后定义了一个 QPen 类的对象 pen,设置其线宽、颜色、线型等,然后设置为 painter 的 pen。再定义了一个 QBrush 类的对象 brush,设置其颜色为黄色,填充方式为实体填充,然后设置

为 painter 的 brush。

这样设置好 painter 的 pen 和 brush 属性后,调用 QPainter 类的 drawRect()函数,就可以绘制前面定义的矩形了,矩形框的线条特性由 pen 决定,填充特性由 brush 决定。运行程序就可以得到如图 8-1 所示的居于界面中间的填充矩形框。

为了不使程序结构太过于复杂,在 paintEvent()函数里直接设置 pen 和 brush 的各种属性,而不是设计复杂的界面程序来修改这些设置。要实现什么绘图功能,只需在 paintEvent()函数里直接修改代码即可。

## 8.1.2 QPen 的主要功能

QPen 用于绘图时对线条进行设置,主要包括线宽、颜色、线型等,表 8-1 是 QPen 类的主要接口函数。通常一个设置函数都有一个对应的读取函数,例如 setColor()用于设置画笔颜色,对应的读取画笔颜色的函数为 color(),表 8-1 仅列出设置函数(省略了函数参数中的 const 关键字)。

表 8-1 QPen 的主要函数

函数原型	功能
void setColor(QColor &color)	设置画笔颜色,即线条颜色
void setWidth(int width)	设置线条宽度
void setStyle(Qt::PenStyle style)	设置线条样式,参数为 Qt::PenStyle 枚举类型
void setCapStyle(Qt::PenCapStyle style)	设置线条端点样式,参数为 Qt::PenCapStyle 枚举类型
void setJoinStyle(Qt::PenJoinStyle style)	设置连接样式,参数为 Qt::PenJoinStyle 枚举类型

线条颜色和宽度的设置无需多说,QPen 影响线条特性的另外 3 个主要属性是线条样式(style)、端点样式(capStyle)和连接样式(joinStyle)。

### 1. 线条样式

setStyle(Qt::PenStyle style)函数用于设置线条样式,参数是一个枚举类型 Qt::PenStyle 的常量,几种典型的线条样式的绘图效果如图 8-2 所示。Qt::PenStyle 类型还有一个常量 Qt::NoPen 表示不绘制线条。

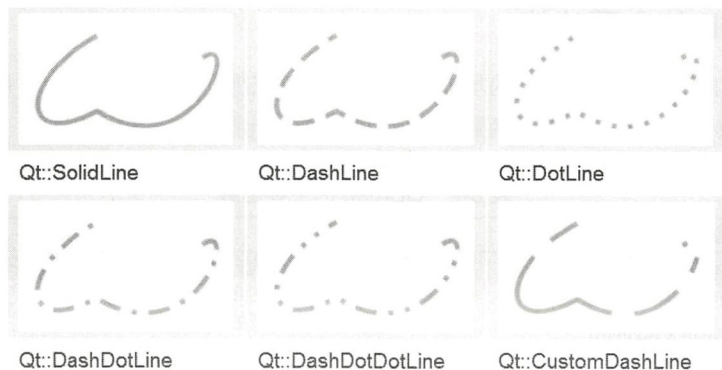


图 8-2 各种样式的线条(来自 Qt 帮助文件)

除了几种基本的线条样式外,用户还可以自定义线条样式,自定义线条样式时需要用到



setDashOffset()和 setDashPattern()函数。

2. 线条端点样式

setCapStyle (Qt::PenCapStyle style)函数用于设置线条端点样式，参数是一个枚举类型 Qt::PenCapStyle 的常量，该枚举类型的 3 种取值及其绘图效果如图 8-3 所示。



图 8-3 各种线条端点样式（来自 Qt 帮助文件）

3. 线条连接样式

setJoinStyle (Qt::PenJoinStyle style)函数用于设置线条连接样式，参数是一个枚举类型 Qt::PenJoinStyle 的常量，该枚举类型的取值及其绘图效果如图 8-4 所示。

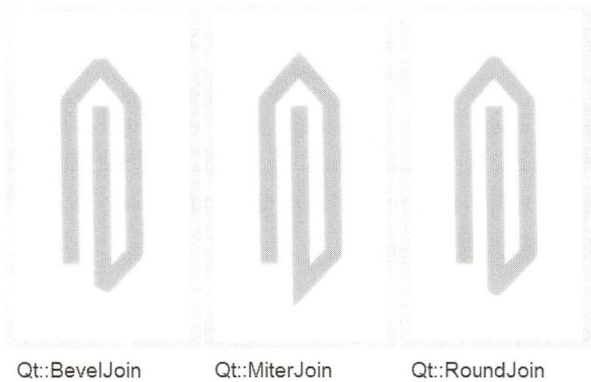


图 8-4 各种线条连接样式（来自 Qt 帮助文件）

8.1.3 QBrush 的主要功能

QBrush 定义了 QPainter 绘图时的填充特性，包括填充颜色、填充样式、材质填充时的材质图片等，其主要函数见表 8-2（省略了函数参数中的 const 关键字）。

表 8-2 QBrush 的主要函数

函数原型	功能
void setColor(QColor &color)	设置画刷颜色，实体填充时即为填充颜色
void setStyle(Qt::BrushStyle style)	设置画刷样式，参数为 Qt::BrushStyle 枚举类型
void setTexture(QPixmap &pixmap)	设置一个 QPixmap 类型的图片作为画刷的图片，画刷样式自动设置为 Qt::TexturePattern
void setTextureImage(QImage &image)	设置一个 QImage 类型的图片作为画刷的图片，画刷样式自动设置为 Qt::TexturePattern

setStyle(Qt::BrushStyle style)函数设置画刷的样式，参数是 Qt::BrushStyle style 枚举类型，该枚

枚举类型典型的几种取值见表 8-3，详细的取值请参考 Qt 的帮助文件。几种典型取值的填充效果如图 8-5 所示。

表 8-3 枚举类型 Qt:: BrushStyle 几个主要常量及其意义

枚举常量	描述
Qt:: NoBrush	不填充
Qt:: SolidPattern	单一颜色填充
Qt:: HorPattern	水平线填充
Qt:: VerPattern	垂直线填充
Qt:: LinearGradientPattern	线性渐变，需要使用 QLinearGradient 类对象作为 Brush
Qt:: RadialGradientPattern	辐射渐变，需要使用 QRadialGradient 类对象作为 Brush
Qt:: ConicalGradientPattern	圆锥型渐变，需要使用 QConicalGradient 类对象作为 Brush
Qt::TexturePattern	材质填充，需要指定 texture 或 textureImage 图片

渐变填充需要使用专门的类作为 Brush 赋值给 QPainter，这部分在后面详细介绍。其他各种线型填充只需设置类型参数即可，使用材质需要设置材质图片。

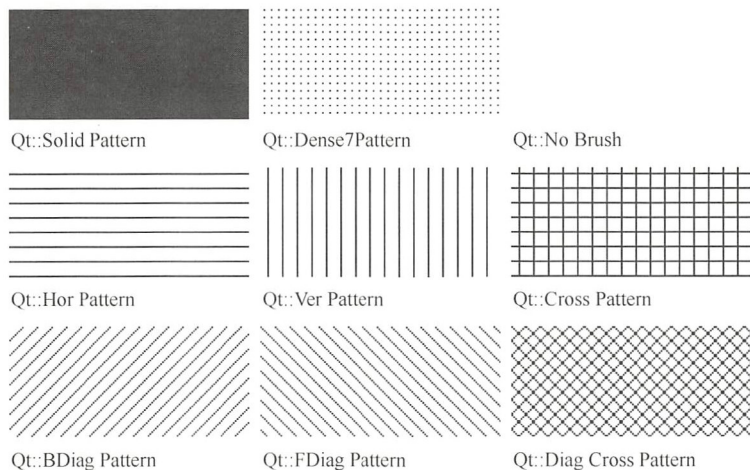


图 8-5 Qt::BrushStyle 几种填充样式（来自 Qt 帮助文件）

下面是使用资源文件里的一个图片进行材质填充的示例程序，用材质图片填充一个矩形，程序运行结果如图 8-6 所示。

```
void Widget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    int W=this->width(); //绘图区宽度
    int H=this->height(); //绘图区高度
    QRect rect(W/4,H/4,W/2,H/2); //中间区域矩形框
    //设置画笔
    QPen pen;
    pen.setWidth(3); //线宽
    pen.setColor(Qt::red); //划线颜色
    pen.setStyle(Qt::SolidLine); //线的类型，实线、虚线等
    painter.setPen(pen);
    //设置画刷
```

```

QPixmap texturePixmap(":/images/images/texture.jpg");
QBrush brush;
brush.setStyle(Qt::TexturePattern); //画刷填充样式
brush.setTexture(texturePixmap); //设置材质图片
painter.setBrush(brush);
//绘图
painter.drawRect(rect);
}

```

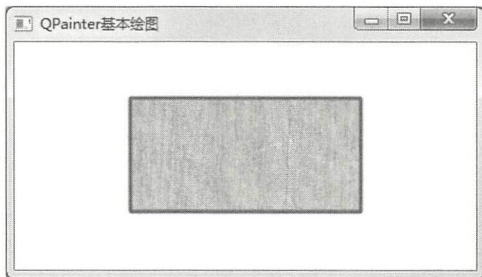


图 8-6 材质填充效果

### 8.1.4 渐变填充

使用渐变填充需要用渐变类的对象作为 Painter 的 brush，有 3 个实现渐变填充的类。

- **QLinearGradient**: 线性渐变。指定一个起点及其颜色，终点及其颜色，还可以指定中间的某个点的颜色，起点至终点之间的颜色会线性插值计算，得到线性渐变的填充颜色。
- **QRadialGradient**: 有简单辐射渐变和扩展辐射渐变两种方式。简单辐射渐变是在一个圆内的一个焦点和一个端点之间生成渐变颜色，扩展辐射渐变是在一个焦点圆和一个中心圆之间生成渐变色。
- **QConicalGradient**: 圆锥形渐变，围绕一个中心点逆时针生成渐变颜色。

这 3 种渐变的示例效果如图 8-7 所示。



图 8-7 3 种渐变填充的效果 (来自 Qt 帮助文件): (左) QLinearGradient; (中) QRadialGradient; (右) QConicalGradient

这 3 个渐变类都继承自 QGradient 类，除了生成渐变颜色的方式不同之外，在设定的渐变颜色坐标范围之外，还需要用 QGradient 类的 `setSpread(QGradient::Spread method)` 函数设置延展方式。枚举类型 `QGradient::Spread` 有 3 种取值，分别表示 3 种延展效果，图 8-8 是使用辐射渐变时 3 种延展方式的效果。`setSpread()` 对圆锥形渐变不起作用。

- **PadSpread** 模式是用结束点的颜色填充外部区域，这是缺省的方式。
- **RepeatSpread** 模式是重复使用渐变方式填充外部区域。
- **ReflectSpread** 是反射式重复使用渐变方式填充外部区域。

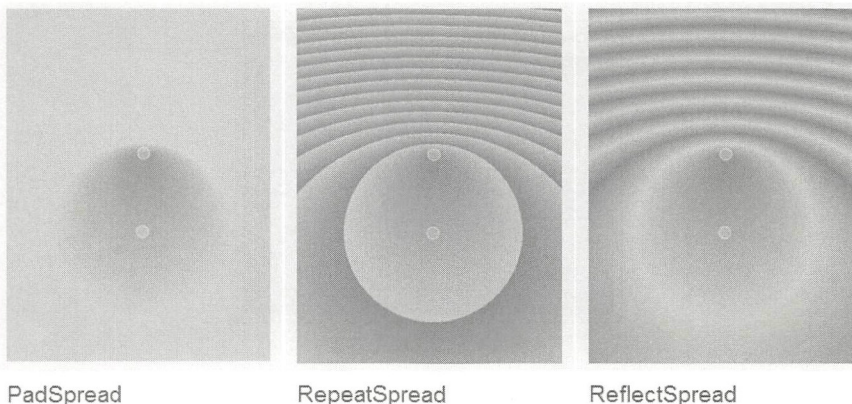


图 8-8 3 种渐变延展效果 (来自 Qt 帮助文件)

下面的代码演示使用渐变效果绘图，程序中使用了辐射渐变。

```
void Widget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    int W=this->width();
    int H=this->height();
    //径向渐变
    QRadialGradient radialGrad(W/2,H/2,qMax(W/8,H/8),W/2,H/2);
    radialGrad.setColorAt(0,Qt::green);
    radialGrad.setColorAt(1,Qt::blue);
    radialGrad.setSpread(QGradient::ReflectSpread);
    painter.setBrush(radialGrad);
    //绘图
    painter.drawRect(this->rect()); //填充更大区域，会有延展效果
}
```

上面的代码中定义 QRadialGradient 对象时使用的构造函数原型是：

```
QRadialGradient(qreal cx, qreal cy, qreal radius, qreal fx, qreal fy)
```

其中,  $(cx, cy)$  是辐射填充的中心点, 程序中设置为  $(W/2, H/2)$ , 也就是 Widget 窗口的中心;  $radius$  是辐射填充区的半径, 程序中设置为  $qMax(W/8, H/8)$ ;  $(fx, fy)$  是焦点坐标, 程序中设置为  $(W/2, H/2)$ , 与中心点相同。

设置辐射渐变的起点颜色和终点颜色的语句是：

```
radialGrad.setColorAt(0,Qt::green);
radialGrad.setColorAt(1,Qt::blue);
```

这里的“点”使用了逻辑坐标, 0 表示起点, 即辐射中心点; 1 表示终点, 即填充区圆的圆周。再用 `setSpread()` 函数设置延展方式为 `QGradient::ReflectSpread`。

最后的绘图语句是：

```
painter.drawRect(this->rect());
```

这里绘制一个矩形, 但是使用的矩形是 `this->rect()`, 即 Widget 窗口的整个矩形区域, 它大于定义的辐射填充区域, 所以会有延展效果。程序运行效果如图 8-9 所示。



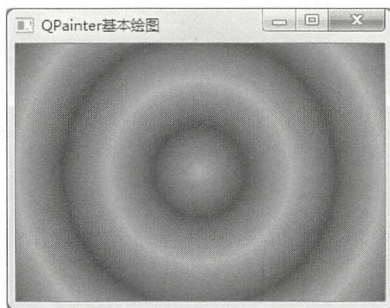


图 8-9 辐射填充效果

若使用线性渐变填充，示例代码如下（只列出 `QLinearGradient` 定义的部分）。定义 `QLinearGradient` 对象时指定了线性渐变的起点和终点，设置颜色时可以在起点和终点之间设置多个点的颜色值。

```
// QLinearGradient linearGrad(rect.left(),rect.top(),
//                               rect.right(),rect.bottom()); //对角线
QLinearGradient linearGrad(rect.left(), rect.top(), rect.right(), rect.top()); //
从左到右
linearGrad.setColorAt(0,Qt::blue); //起点颜色
linearGrad.setColorAt(0.5,Qt::green); //中间点颜色
linearGrad.setColorAt(1,Qt::red); //终点颜色
linearGrad.setSpread(QGradient::ReflectSpread); //展布模式
painter.setBrush(linearGrad);
```

创建 `QLinearGradient` 对象时传递了两个坐标点，分别表示填充区的起点和终点，起点和终点的定义方式不同，可以实现水平渐变、垂直渐变、或对角渐变等不同效果。

使用圆锥形渐变的示例代码如下。创建 `QConicalGradient` 对象时指定了中心点坐标和起始角度，然后设置多个点的颜色。但是注意，圆锥形填充没有延展效果。

```
QConicalGradient conigrad(W/2,H/2,45);
conigrad.setColorAt(0,Qt::yellow);
conigrad.setColorAt(0.5,Qt::blue);
conigrad.setColorAt(1,Qt::green);
painter.setBrush(conigrad);
```

## 8.1.5 QPainter 绘制基本图形元件



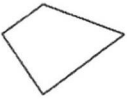
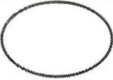





### 1. 基本图形元件

`QPainter` 提供了很多绘制基本图形的功能，包括点、直线、椭圆、矩形、曲线等，由这些基本的图形可以构成复杂的图形。`QPainter` 中提供的绘制基本图元的函数见表 8-4。每个函数基本上都有多种参数形式，这里只列出函数名，给出了其中一种参数形式的示例代码，并且假设已经通过以下的代码获得了绘图窗口的 `painter`、窗口宽度 `W` 和高度 `H`。

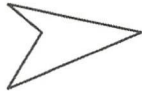
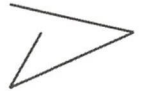




```
QPainter painter(this);
int W=this->width(); //绘图区宽度
int H=this->height(); //绘图区高度
```

同一个函数名的其他参数形式的函数原型可查阅 Qt 的帮助文件。

表 8-4 QPainter 绘制基本图形元件的函数

函数名	功能和示例代码	示例图形
drawArc	画弧线, 例如 <pre>QRect rect(W/4,H/4,W/2,H/2); int startAngle = 90 * 16;    //起始 90° int spanAngle = 90 * 16;    //旋转 90° painter.drawArc(rect, startAngle, spanAngle);</pre>	
drawChord	画一段弦, 例如 <pre>QRect rect(W/4,H/4,W/2,H/2); int startAngle = 90 * 16;    //起始 90° int spanAngle = 90 * 16;    //旋转 90° painter.drawChord(rect, startAngle, spanAngle);</pre>	
drawConvexPolygon	根据给定的点画凸多边形 <pre>QPoint points[4]={     QPoint(5*W/12,H/4),     QPoint(3*W/4,5*H/12),     QPoint(5*W/12,3*H/4),     QPoint(W/4,5*H/12), }; painter.drawConvexPolygon(points, 4);</pre>	
drawEllipse	画椭圆 <pre>QRect rect(W/4,H/4,W/2,H/2); painter.drawEllipse(rect);</pre>	
drawImage	在指定的矩形区域内绘制图片 <pre>QRect rect(W/4,H/4,W/2,H/2); QImage image(":/images/images/qt.jpg"); painter.drawImage(rect, image);</pre>	
drawLine	画直线 <pre>QLine Line(W/4,H/4,W/2,H/2); painter.drawLine(Line);</pre>	
drawLines	画一批直线 <pre>QRect rect(W/4,H/4,W/2,H/2); QVector&lt;QLine&gt; Lines; Lines.append(QLine(rect.topLeft(),rect.bottomRight())); Lines.append(QLine(rect.topRight(),rect.bottomLeft())); Lines.append(QLine(rect.topLeft(),rect.bottomLeft())); Lines.append(QLine(rect.topRight(),rect.bottomRight())); painter.drawLines(Lines);</pre>	
drawPath	绘制由 QPainterPath 对象定义的路线 <pre>QRect rect(W/4,H/4,W/2,H/2); QPainterPath path; path.addEllipse(rect); path.addRect(rect); painter.drawPath(path);</pre>	
drawPie	绘制扇形 <pre>QRect rect(W/4,H/4,W/2,H/2); int startAngle = 40 * 16; //起始 40° int spanAngle = 120 * 16; //旋转 120° painter.drawPie(rect, startAngle, spanAngle);</pre>	
drawPixmap	绘制 QPixmap 图片 <pre>QRect rect(W/4,H/4,W/2,H/2); QPixmap pixmap(":/images/images/qt.jpg"); painter.drawPixmap(rect, pixmap);</pre>	
drawPoint	画一个点 <pre>painter.drawPoint(QPoint(W/2,H/2));</pre>	

续表

函数名	功能和示例代码	示例图形
	画一批点	
drawPoints	<pre> QPoint points[]={     QPoint(5*W/12,H/4),     QPoint(3*W/4,5*H/12),     QPoint(2*W/4,5*H/12) }; painter.drawPoints(points, 3); </pre>	
	画多边形, 最后一个点会和第一个点闭合	
drawPolygon	<pre> QPoint points[]={     QPoint(5*W/12,H/4),     QPoint(3*W/4,5*H/12),     QPoint(5*W/12,3*H/4),     QPoint(2*W/4,5*H/12) }; painter.drawPolygon(points, 4); </pre>	
	画多点连接的线, 最后一个点不会和第一个点连接	
drawPolyline	<pre> QPoint points[]={     QPoint(5*W/12,H/4),     QPoint(3*W/4,5*H/12),     QPoint(5*W/12,3*H/4),     QPoint(2*W/4,5*H/12), }; painter.drawPolyline(points, 4); </pre>	
	画矩形	
drawRect	<pre> QRect rect(W/4,H/4,W/2,H/2); painter.drawRect(rect); </pre>	
	画圆角矩形	
drawRoundedRect	<pre> QRect rect(W/4,H/4,W/2,H/2); painter.drawRoundedRect(rect,20,20); </pre>	
	绘制文本, 只能绘制单行文字, 字体的大小等属性由 QPainter::font()决定。	
drawText	<pre> QRect rect(W/4,H/4,W/2,H/2); QFont font; font.setPointSize(30); font.setBold(true); painter.setFont(font); painter.drawText(rect,"Hello,Qt"); </pre>	
	擦除某个矩形区域, 等效于用背景色填充该区域	
eraseRect	<pre> QRect rect(W/4,H/4,W/2,H/2); painter.eraseRect(rect); </pre>	
	填充某个 QPainterPath 定义的绘图路径, 但是轮廓线不显示	
fillPath	<pre> QRect rect(W/4,H/4,W/2,H/2); QPainterPath path; path.addEllipse(rect); path.addRect(rect); painter.fillPath(path,Qt::red); </pre>	
	填充一个矩形, 无边框线	
fillRect	<pre> QRect rect(W/4,H/4,W/2,H/2); painter.fillRect(rect,Qt::green); </pre>	

这些基本图形元件的绘制用户可以通过修改 samp8\_1 的 paintEvent() 里的代码进行测试, 这里就不再详细举例和说明了。

## 2. QPainterPath 的使用

在表 8-4 列举的 QPainter 绘制基本图形元件的函数中, 一般的图形元件的绘制都比较简单和

直观, 只有 `drawPath()` 函数是绘制一个复合的图形对象, 它使用一个 `QPainterPath` 类型的参数作为绘图对象。`drawPath()` 函数的原型是:

```
void QPainter::drawPath(const QPainterPath &path)
```

`QPainterPath` 是一系列绘图操作的顺序集合, 便于重复使用。一个 `PainterPath` 由许多基本的绘图操作组成, 如绘图点移动、划线、画圆、画矩形等, 一个闭合的 `PainterPath` 是终点和起点连接起来的绘图路径。使用 `QPainterPath` 的优点是绘制某些复杂形状时只需创建一个 `PainterPath`, 然后调用 `QPainter::drawPath()` 就可以重复使用。例如绘制一个复杂的星星图案需要多次调用 `lineto()` 函数, 定义一个 `QPainterPath` 类型的变量 `path` 记录这些绘制过程, 再调用 `drawPath(path)` 就可以完成星型图案的绘制。

`QPainterPath` 提供了很多函数可以添加各种基本图形元件的绘制, 其功能与 `QPainter` 提供的绘制基本图件的功能类似, 也有一些用于 `PainterPath` 的专用函数, 如 `closeSubpath()`、`connectPath()` 等, 对于 `QPainterPath` 的函数功能不做详细说明, 可以参考 Qt 帮助文件查看 `QPainterPath` 类的详细描述。在下一节的实例 `samp8_2` 中将结合 `QPainter` 的坐标变换功能演示 `QPainterPath` 绘制多个星星的实现方法。

## 8.2 坐标系统和坐标变换

### 8.2.1 坐标变换函数

`QPainter` 在窗口上绘图的默认坐标系如图 8-1 所示, 这是绘图设备的物理坐标。为了绘图的方便, `QPainter` 提供了一些坐标变换的功能, 通过平移、旋转等坐标变换, 得到一个逻辑坐标系, 使用逻辑坐标系在某些时候绘图更方便。坐标变换函数见表 8-5。

表 8-5 QPainter 有关坐标变换操作的函数

分组	函数原型	功能
坐标变换	<code>void translate(qreal dx, qreal dy)</code>	坐标系统平移一定的偏移量, 坐标原点平移到新的点
	<code>void rotate(qreal angle)</code>	坐标系统顺时针旋转一个角度
	<code>void scale(qreal sx, qreal sy)</code>	坐标系统缩放
	<code>void shear(qreal sh, qreal sv)</code>	坐标系统做扭转变换
状态保存与恢复	<code>void save()</code>	保存 painter 当前的状态, 就是将当前状态压入堆栈
	<code>void restore()</code>	恢复上一次状态, 就是从堆栈中弹出上次状态
	<code>void resetTransform()</code>	复位所有的坐标变换

常用的坐标变换是平移、旋转和缩放, 使用世界坐标变换矩阵也可以实现这些变换功能, 但是需要单独定义一个 `QTransform` 类的变量, 对于 `QPainter` 来说, 简单的坐标变换使用 `QPainter` 自有的坐标变换函数就足够了。

#### 1. 坐标平移

坐标平移函数是 `translate()`, 其中一种参数形式的函数原型是:

```
void translate(qreal dx, qreal dy)
```

表示将坐标系统水平方向平移 `dx` 个单位, 垂直方向平移 `dy` 个单位, 在缺省的坐标系统中,



单位就是像素。如果是从原始状态平移  $(dx, dy)$ ，那么平移后的坐标原点就移到了  $(dx, dy)$ 。

假设一个绘图窗口宽度为 300 像素，高度为 200 像素，则其原始坐标系统如图 8-10 左所示；若执行平移函数 `translate(150, 100)`，则坐标系统水平向右平移 150 像素，向下平移 100 像素，平移后的坐标系统如图 8-10 右所示，坐标原点在窗口的中心，而左上角的坐标变为  $(-150, -100)$ ，右下角的坐标变为  $(150, 100)$ 。如此将坐标原点变换到窗口中心在绘制某些图形时是非常方便的。

## 2. 坐标旋转

坐标旋转的函数是 `rotate()`，其函数原型为：

```
void rotate(qreal angle)
```

它是将坐标系统绕坐标原点顺时针旋转 `angle` 角度，单位是度。当 `angle` 为正数时是顺时针旋转，为负数时是逆时针旋转。

在图 8-10 右的基础上，若执行 `rotate(90)`，则得到图 8-11 所示的坐标系统。

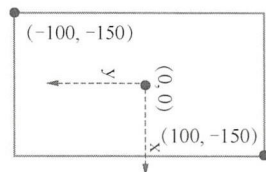
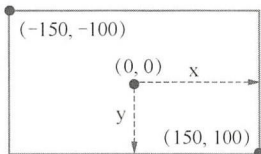
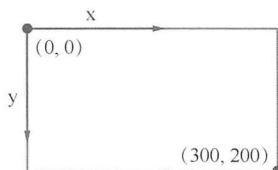


图 8-10 (左) 原始坐标系统；(右) 平移  $(150, 100)$  后的坐标系统 图 8-11 对图 8-10 右图旋转  $90^\circ$  之后的坐标系

**注意** 旋转之后并不改变窗口矩形的实际大小，只是改变了坐标轴的方向。

在图 8-11 的新坐标系下，窗口左上角的坐标变成了  $(-100, 150)$ ，而右下角的坐标变成了  $(100, -150)$ 。

## 3. 缩放

缩放函数是 `scale()`，其函数原型为：

```
void scale(qreal sx, qreal sy)
```

其中，`sx`, `sy` 分别为横向和纵向缩放比例，比例大于 1 是放大，小于 1 是缩小。

## 4. 状态保存与恢复

进行坐标变换时，`QPainter` 内部实际上有一个坐标变换矩阵，用 `save()` 保存当前坐标状态，用 `restore()` 恢复上次保存的坐标状态，这两个函数必须配对使用，操作的是一个堆栈对象。

`resetTransform()` 函数则是复位所有坐标变换操作，恢复原始的坐标系统。

## 8.2.2 坐标变换绘图实例

### 1. 绘制 3 个五角星的程序

创建一个基于 `QWidget` 的窗口的应用程序 `samp8_2`，窗体上不放置任何组件。在 `Widget` 类的构造函数和 `paintEvent()` 事件中编写代码，代码如下。

```
Widget::Widget(QWidget *parent) :    QWidget(parent),    ui(new Ui::Widget)
{
    ui->setupUi(this);
    setPalette(QPalette(Qt::white)); //设置窗口背景色
```

```

        setAutoFillBackground(true);
        resize(600,300); //固定初始化窗口大小
    }
void Widget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing);
    painter.setRenderHint(QPainter::TextAntialiasing);
    //生成五角星的5个顶点的坐标,假设原点在五角星中心
    qreal R=100; //半径
    const qreal Pi=3.14159;
    qreal deg=Pi*72/180;
    QPoint points[5]={
        QPoint(R,0),
        QPoint(R*std::cos(deg),-R*std::sin(deg)),
        QPoint(R*std::cos(2*deg),-R*std::sin(2*deg)),
        QPoint(R*std::cos(3*deg),-R*std::sin(3*deg)),
        QPoint(R*std::cos(4*deg),-R*std::sin(4*deg)),
    };
    //设置字体
    QFont font;
    font.setPointSize(12);
    font.setBold(true);
    painter.setFont(font);
    //设置画笔
    QPen penLine;
    penLine.setWidth(2); //线宽
    penLine.setColor(Qt::blue); //划线颜色
    penLine.setStyle(Qt::SolidLine); //线的类型
    penLine.setCapStyle(Qt::FlatCap); //线端点样式
    penLine.setJoinStyle(Qt::BevelJoin); //线的连接点样式
    painter.setPen(penLine);
    //设置画刷
    QBrush brush;
    brush.setColor(Qt::yellow); //画刷颜色
    brush.setStyle(Qt::SolidPattern); //画刷填充样式
    painter.setBrush(brush);
    //设计绘制五角星的 PainterPath, 以便重复使用
    QPainterPath starPath;
    starPath.moveTo(points[0]);
    starPath.lineTo(points[2]);
    starPath.lineTo(points[4]);
    starPath.lineTo(points[1]);
    starPath.lineTo(points[3]);
    starPath.closeSubpath(); //闭合路径, 最后一个点与第一个点相连
    starPath.addText(points[0],font,"0"); //显示端点编号
    starPath.addText(points[1],font,"1");
    starPath.addText(points[2],font,"2");
    starPath.addText(points[3],font,"3");
    starPath.addText(points[4],font,"4");
    //绘图
    painter.save(); //保存坐标状态
    painter.translate(100,120); //平移
    painter.drawPath(starPath); //画星星
    painter.drawText(0,0,"S1");
}

```

```

painter.restore(); //恢复坐标状态

painter.translate(300,120); //平移
painter.scale(0.8,0.8); //缩放
painter.rotate(90); //顺时针旋转 90 度
painter.drawPath(starPath); //画星星
painter.drawText(0,0,"S2");

painter.resetTransform(); //复位所有坐标变换
painter.translate(500,120); //平移
painter.rotate(-145); //逆时针旋转 145 度
painter.drawPath(starPath); //画星星
painter.drawText(0,0,"S3");
}

```

运行该实例程序，得到如图 8-12 所示的结果，在窗口上绘制了 3 个五角星。

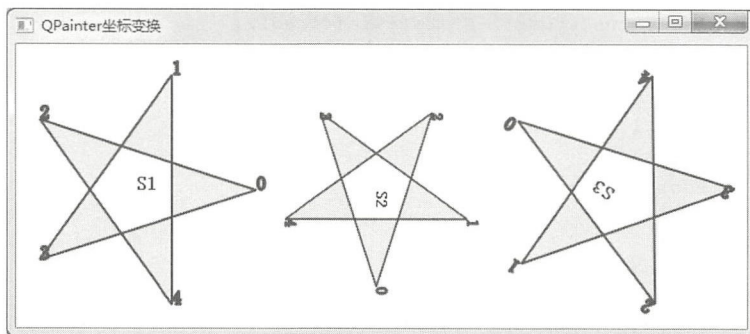


图 8-12 使用 QPainterPath 和坐标变换的绘图效果

第 1 个是原始的五角星；第 2 个是缩小为 0.8 倍，顺时针旋转 90 度的五角星；第 3 个是逆时针旋转 145 度的五角星。这个程序中使用到了 QPainterPath 和 QPainter 的坐标变换功能。

## 2. 绘制五角星的 QPainterPath 的定义

首先假设一个五角星的中心点是原点，第 0 个点在 X 轴上，五角星外接圆半径为 100，计算出 5 个点的坐标，保存在 points 数组中。

然后定义了一个 QPainterPath 类的变量 starPath，用于记录画五角星的过程，就是几个点的连线过程，并且标注点的编号。使用 QPainterPath 的优点就是定义一个 QPainterPath 类型的变量记录一个复杂图形的绘制过程后，可以重复使用。虽然 points 数组中的点的坐标是假设五角星的中心点是原点，在绘制不同的五角星时只需将坐标平移到新的原点位置，就可以绘制不同的五角星。

绘制第 1 个五角星的程序是：

```

painter.save(); //保存坐标状态
painter.translate(100,120);
painter.drawPath(starPath); //画星星
painter.drawText(0,0,"S1");
painter.restore(); //恢复坐标状态

```

这里，save() 函数保存当前坐标状态（也就是坐标的原始状态），然后将坐标原点平移到 (100,120)，调用绘制路径的函数 drawPath(starPath) 绘制五角星，在五角星的中心标注“S1”表示第 1 个五角星，

最后调用 `restore()` 函数恢复上次的坐标状态。这样就以 (100,120) 为中心点绘制了第 1 个五角星。

绘制第 2 个五角星的程序是：

```
painter.translate(300,120); //平移
painter.scale(0.8,0.8); //缩放
painter.rotate(90); //顺时针旋转 90 度
painter.drawPath(starPath); //画星星
painter.drawText(0,0,"S2");
```

这里首先调用坐标平移函数 `translate(300, 120)`。由于上次 `restore()` 之后回到坐标初始状态，所以这次平移后，坐标原点到了物理坐标 (300, 120)。而如果没有上一个 `restore()`，会在上一次的坐标基础上平移。

绘图之前调用了缩放函数 `scale(0.8, 0.8)`，使得缩小到原来的 0.8，再顺时针旋转  $90^\circ$ ，然后用绘制路径函数 `drawPath(starPath)` 绘制五角星，就得到了第 2 个五角星。

绘制第 3 个五角星时首先使坐标复位，即：

```
painter.resetTransform(); //复位所有坐标变换
```

这样会复位所有坐标变换，又回到了原始坐标。

## 8.2.3 视口和窗口

### 1. 视口和窗口的定义与原理

绘图设备的物理坐标是基本的坐标系，通过 `QPainter` 的平移、旋转等变换可以得到更容易操作的逻辑坐标。

为了实现更方便的坐标，`QPainter` 还提供了视口 (Viewport) 和窗口 (Window) 坐标系，通过 `QPainter` 内部的坐标变换矩阵自动转换为绘图设备的物理坐标。

视口表示绘图设备的任意一个矩形区域的物理坐标，可以只选取物理坐标的一个矩形区域用于绘图。默认情况下，视口等于绘图设备的整个矩形区。

窗口与视口是同一个矩形，只不过是用逻辑坐标定义的坐标系。窗口可以直接定义矩形区的逻辑坐标范围。图 8-13 是对视口和窗口的图示说明。

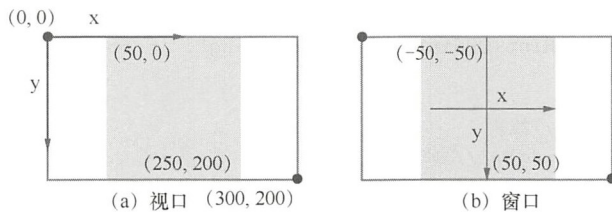


图 8-13 视口和窗口示意图

图 8-13 左图中的矩形框代表绘图设备的物理大小和坐标范围，假设宽度为 300 像素，高度为 200 像素。现在要取其中的一个正方形区域作为视口，灰色的正方形就是视口，绘图设备的物理坐标中，视口的左上角坐标为 (50, 0)，右下角坐标为 (250, 200)。定义此视口，可以使用 `QPainter` 的 `setViewport()` 函数，其函数原型为：



```
void QPainter::setViewport(int x, int y, int width, int height)
```

要定义图 8-13 左图中的视口, 使用下面的语句:

```
painter.setViewport(50,0,200,200);
```

表示从绘图设备物理坐标系统的起点(50, 0)开始, 取宽度为 200、高度为 200 的一个矩形区域作为视口。

对于图 8-13 左图的视口所表示的正方形区域, 定义一个窗口(图 8-13 右图), 窗口坐标的中心在正方形中心, 并设置正方形的逻辑边长为 100。可使用 QPainter 的 setWindow()函数, 其函数原型为:

```
void QPainter::setWindow(int x, int y, int width, int height)
```

所以, 此处定义窗口的语句是:

```
painter.setWindow(-50,-50,100,100);
```

它表示对应于视口的矩形区域, 其窗口左上角的逻辑坐标是(-50, -50), 窗口宽度为 100, 高度为 100。这里设置的窗口还是一个正方形, 使得从视口到窗口变换时, 长和宽的变化比例是相同的。实际可以任意指定窗口的逻辑坐标范围, 长和宽的变化比例不相同也是可以的。

## 2. 视口和窗口的使用实例

使用窗口坐标的优点是, 只需按照窗口坐标定义来绘图, 而不用管实际的物理坐标范围的大小。例如在一个固定边长为 100 的正方形窗口内绘图, 当实际绘图设备大小变化时, 绘制的图形会自动变化大小。这样, 就可以将绘图功能与绘图设备隔离开来, 使得绘图功能适用于不同大小、不同类型的设备。

实例 samp8\_3 演示了使用视口和窗口的方法, 项目创建与 samp8\_1 类似, 只在 Widget 的 paintEvent()事件里添加绘图代码。

```
void Widget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    int W=width();
    int H=height();
    int side=qMin(W,H); //取长和宽的较小值
    QRect rect((W-side)/2, (H-side)/2, side, side); //viewport 矩形区
    painter.drawRect(rect); //Viewport 的矩形区域
    painter.setViewport(rect); //设置 Viewport
    painter.setWindow(-100,-100,200,200); // 设置窗口大小, 逻辑坐标
    painter.setRenderHint(QPainter::Antialiasing);
    //设置画笔
    QPen pen;
    pen.setWidth(1); //线宽
    pen.setColor(Qt::red); //划线颜色
    pen.setStyle(Qt::SolidLine); //线的类型
    painter.setPen(pen);
    for(int i=0; i<36;i++)
    {
        painter.drawEllipse(QPoint(50,0),50,50);
        painter.rotate(10);
    }
}
```

运行实例程序 samp8\_3, 可以得到如图 8-14 所示的图形效果。当窗口的宽度大于高度时, 以

高度作为正方形的边长；当高度大于宽度时，以宽度作为正方形边长，且图形是自动缩放的。

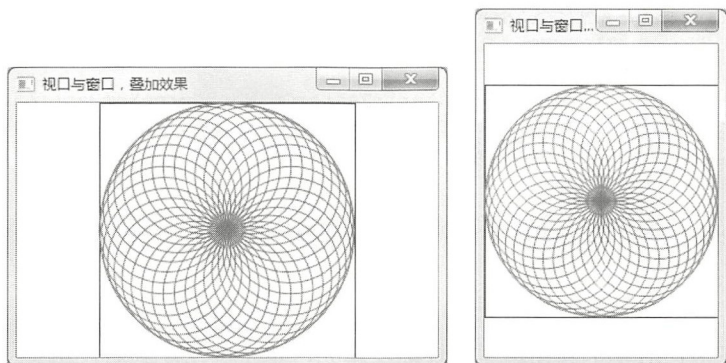


图 8-14 实例 samp8\_3 使用窗口坐标的绘图效果

程序首先定义了一个正方形视口，正方形以绘图设备的长、宽中较小者为边长。

然后定义窗口，定义的窗口是原点在中心，边长为 200 的正方形。

图 8-14 的图形效果实际上是画了 36 个圆得到的，循环部分的代码如下：

```
for(int i=0; i<36;i++)
{
    painter.drawEllipse(QPoint(50,0),50,50);
    painter.rotate(10);
}
```

每个圆的圆心在 X 轴上的(50, 0)，半径为 50。画完一个圆之后坐标系旋转 10°，再画相同的圆，这就巧妙应用了坐标轴的旋转。

## 8.2.4 绘图叠加的效果

对上面的程序稍作修改，增加渐变填充和叠加效果的设置。

```
void Widget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    int W=width();
    int H=height();
    int side=qMin(W,H); //取长和宽的较小值
    QRect rect((W-side)/2, (H-side)/2,side,side); //viewport 矩形区
    painter.drawRect(rect); //Viewport 大小
    painter.setViewport(rect); //设置 Viewport
    painter.setWindow(-100,-100,200,200); // 设置窗口大小，逻辑坐标
    painter.setRenderHint(QPainter::Antialiasing);
    //设置画笔
    QPen pen;
    pen.setWidth(1); //线宽
    pen.setColor(Qt::red); //划线颜色
    pen.setStyle(Qt::SolidLine); //线的类型
    painter.setPen(pen);
    //线性渐变
    QLinearGradient linearGrad(0,0,100,0); //从左到右,
```

```

linearGrad.setColorAt(0,Qt::yellow);//起点颜色
linearGrad.setColorAt(1,Qt::green);//终点颜色
linearGrad.setSpread(QGradient::PadSpread); //展布模式
painter.setBrush(linearGrad);
//设置复合模式
painter.setCompositionMode(QPainter::RasterOp_NotSourceXorDestination);
// painter.setCompositionMode(QPainter::CompositionMode_Difference);
// painter.setCompositionMode(QPainter::CompositionMode_Exclusion);
for(int i=0; i<36;i++)
{
    painter.drawEllipse(QPoint(50,0),50,50);
    painter.rotate(10);
}
}

```

在上面的程序中，对单个圆使用了线性渐变填充，单个圆从左到右，由黄色渐变为绿色。

使用 `QPainter::setCompositionMode()` 函数设置组合模式，即后面绘制的图与前面绘制的图的叠加模式。函数参数是一个 `QPainter::CompositionMode` 枚举类型值，可以查看 Qt 帮助，这个枚举类型有近 40 种取值，表示了后绘制图形与前面图形的不同叠加运算方式。

图 8-15 是其中两种叠加模式下的绘图效果，可以发现采用不同的叠加模式，可以得到不同的绘图效果，甚至是意想不到的绚丽效果。用户可以自己修改程序，设置不同渐变颜色、渐变填充模式、不同叠加模式，也许能绘制出更炫的图形呢。

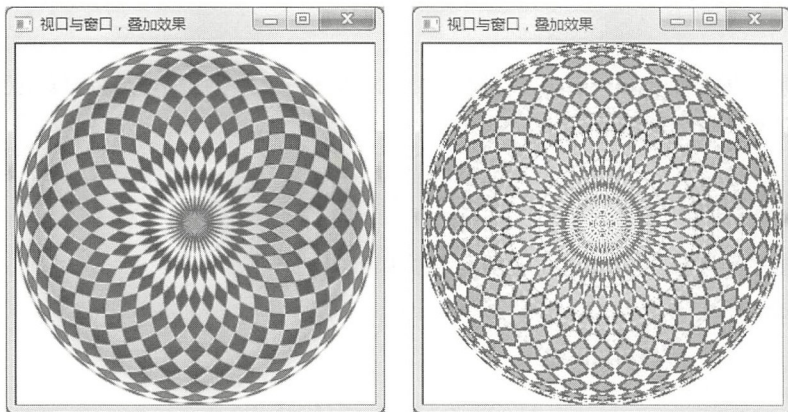


图 8-15 渐变填充和叠加效果，（左）`CompositionMode_Difference` 模式叠加，（右）`RasterOp_NotSourceXorDestination` 模式叠加

## 8.3 Graphics View 绘图架构

### 8.3.1 场景、视图与图形项

采用 `QPainter` 绘图时需要在绘图设备的 `paintEvent()` 事件里编写绘图的程序，实现整个绘图过程。这种方法如同使用 Windows 的画图软件在绘图，绘制的图形是位图，这种方法适合于绘制复杂性不高的固定图形，不能实现图件的选择、编辑、拖放、修改等功能。



Qt 为绘制复杂的可交互图形提供了 Graphics View 绘图架构，是一种基于图形项（Graphics Item）的模型/视图模式，与第 5 章的数据编辑与显示的 Model/View 模式类似。使用 Graphics View 架构可以绘制复杂的有几万个基本图形元件的图形，并且每个图形元件是可选择、可拖放和修改的，类似于矢量绘图软件的绘图功能。

Graphics View 架构主要由 3 个部分组成，即场景、视图和图形项，其构成的 Graphics View 绘图系统结构如图 8-16 所示。

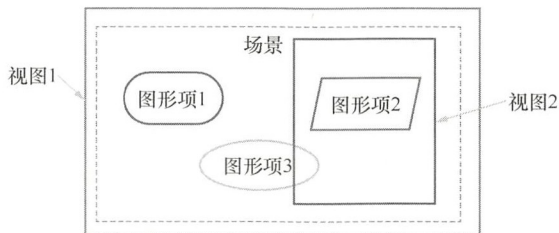


图 8-16 视图、场景、图形项的关系

### 1. 场景

QGraphicsScene 类提供绘图场景（Scene）。场景是不可见的，是一个抽象的管理图形项的容器，可以向场景添加图形项，获取场景中的某个图形项等。场景主要具有如下一些功能：

- 提供管理大量图形项的快速接口；
- 将事件传播给每个图形项；
- 管理每个图形项的状态，例如选择状态、焦点状态等；
- 管理未经变换的渲染功能，主要用于打印。

除了图形项之外，场景还有背景层和前景层，通常由 QBrush 指定，也可以通过重新实现 drawBackground() 和 drawForeground() 函数来实现自定义的背景和前景，实现一些特殊效果。

### 2. 视图

QGraphicsView 提供绘图的视图（View）组件，用于显示场景中的内容。可以为一个场景设置几个视图，用于对同一个数据集提供不同的视图。

在图 8-16 中，虚线框的部分是一个场景，视图 1 比场景大，显示场景的全部内容。缺省情况下，当视图大于场景时，场景在视图的中间部分显示，也可以设置视图的 Alignment 属性控制场景在视图中的显示位置；当视图小于场景时（见图 8-16 中的视图 2），视图只能显示场景的一部分内容，但是会自动提供卷滚条在整个场景内移动。

视图接收键盘和鼠标输入并转换为场景事件，并进行坐标转换后传送给可视场景。

### 3. 图形项

图形项（Graphics Item）就是一些基本的图形元件，图形项的基类是 QGraphicsItem。Qt 提供了一些基本的图形项，如绘制椭圆的 QGraphicsEllipseItem、绘制矩形的 QGraphicsRectItem、绘制文字的 QGraphicsTextItem 等。

QGraphicsItem 支持如下的一些操作：

- 支持鼠标事件响应，包括鼠标按下、移动、释放、双击，还包括鼠标停留、滚轮、快捷菜



单等事件;

- 支持键盘输入, 按键事件;
- 支持拖放操作;
- 支持组合, 可以是父子项关系组合, 也可以是通过 `QGraphicsItemGroup` 类进行组合。

所以, 图形项可以被选择、拖放、组合, 若编写信号槽函数代码, 还可以实现各种编辑功能。一个图形项还可以包含子图形项, 图形项还支持碰撞检测, 即是否与其他图形项碰撞。

在图 8-16 所示的视图、场景和图形项之间的关系示意图中, 场景是图形项的容器, 可以在场景上绘制很多图形项, 每个图形项就是一个对象, 这些图形项可以被选择、拖动等。视图是显示场景的一部分区域的视口, 一个场景可以有多个视图, 一个视图显示场景的部分区域或全部区域, 或从不同角度观察场景。

### 8.3.2 Graphics View 的坐标系统

Graphics View 系统有 3 个有效的坐标系, 图形项坐标、场景坐标、视图坐标。3 个坐标系的示意图如图 8-17 所示。绘图的时候, 场景的坐标等价于 `QPainter` 的逻辑坐标, 一般以场景的中心为原点; 视图坐标与设备坐标相同, 是物理坐标, 缺省以左上角为原点; 图形项坐标是局部逻辑坐标, 一般以图件的中心为原点。

#### 1. 图形项坐标

图形项使用自己的局部坐标 (Item Coordinates), 通常以其中心为 (0, 0), 也是各种坐标变换的中心。图形项的鼠标事件的坐标是用局部坐标表示的, 创建自定义图形项, 绘制图形项时只需考虑其局部坐标, `QGraphicsScene` 和 `QGraphicsView` 会自动进行坐标转换。

一个图形项的位置是其中心点在父坐标系中的坐标, 对于没有父图形项的图形项, 其父对象就是场景, 图形项的位置就是在场景中的坐标。

如果一个图形项还是其他图形项的父项, 父项进行坐标变换时, 子项也做同样的坐标变换。

`QGraphicsItem` 的大多数函数都是在其局部坐标系上操作的, 例如一个图形项的边界矩形 `QGraphicsItem::boundingRect()` 是用局部坐标给出的, 但是 `QGraphicsItem::pos()` 是仅有的几个例外, 它返回的是图形项在父项坐标系中的坐标, 如果是顶层图形项, 就是在场景中的坐标。

#### 2. 视图坐标

视图坐标 (View Coordinates) 就是窗口界面 (widget) 的物理坐标, 单位是像素。视图坐标只与 widget 或视口有关, 而与观察的场景无关。 `QGraphicsView` 视口的左上角坐标总是 (0,0)。

所有的鼠标事件、拖放事件的坐标首先是由视图坐标定义的, 然后用户需要将这些坐标映射为场景坐标, 以便和图形项交互。

#### 3. 场景坐标

场景是所有图形项的基础坐标, 场景坐标 (Scene Coordinates) 描述了每个顶层图形项的位置。创建场景时可以定义场景矩形区的坐标范围, 例如

```
scene=new QGraphicsScene(-400,-300,800,600);
```

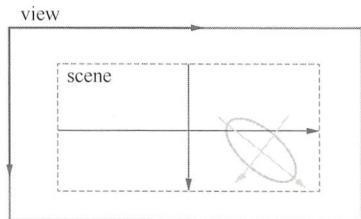


图 8-17 场景、视图、图形项 3 个坐标系之间的关系

这样定义的 scene 是左上角坐标为 (-400, -300)，宽度为 800，高度为 600 的矩形区域，单位是像素。

每个图形项在场景里都有一个位置坐标，由函数 `QGraphicsItem::scenePos()` 给出；还有一个图形项边界矩形，由 `QGraphicsItem::sceneBoundingRect()` 函数给出。边界矩形可以使 `QGraphicsScene` 知道场景的哪个区域发生了变化。场景发生变化时会发射 `QGraphicsScene::changed()` 信号，参数是一个场景的矩形列表，表示发生变化的矩形区。

#### 4. 坐标映射

在场景中操作图形项时，进行场景到图形项、图形项到图形项，或视图到场景之间的坐标变换是比较有用的，即坐标映射 (Coordinate Mapping)。例如，在 `QGraphicsView` 的视口上单击鼠标时，通过函数 `QGraphicsView::mapToScene()` 可以将视图坐标映射为场景坐标，然后用 `QGraphicsScene::itemAt()` 函数可以获取场景中鼠标光标处的图形项。

### 8.3.3 Graphics View 相关的类

Graphics View 结构的主要类包括视图类 `QGraphicsView`、场景类 `QGraphicsScene`，和各种图形项类，图形项类的基类都是 `QGraphicsItem`。

#### 1. QGraphicsView 类的主要接口函数

`QGraphicsView` 是用于观察一个场景的物理窗口，当场景小于视图时，整个场景在视图中可见；当场景大于视图时，视图自动提供卷滚条。

`QGraphicsView` 的视口坐标等于显示设备的物理坐标，但是也可以对 `QGraphicsView` 的坐标进行平移、旋转、缩放等变换。

表 8-6 是 `QGraphicsView` 的主要接口函数。一般的设置函数还有一个对应的读取函数，如 `setScene()` 对应的读取函数是 `scene()`，这里只列出设置函数。并且仅列出函数的返回数据类型，省略了输入参数，函数的详细定义见 Qt 帮助文件。

表 8-6 QGraphicsView 主要函数功能说明

分组	函数	功能描述
场景	<code>void setScene()</code>	设置关联显示的场景
	<code>void setSceneRect()</code>	设置场景在视图中可视的部分的矩形区域
外观	<code>void setAlignment()</code>	设置场景在视图中的对齐方式，缺省是上下都居中
	<code>void setBackgroundBrush()</code>	设置场景的背景画刷
	<code>void setForegroundBrush()</code>	设置场景的前景画刷
	<code>void setRenderHints()</code>	设置视图的绘图选项
交互	<code>void setInteractive()</code>	是否允许场景可交互，如果禁止交互，则任何键盘或鼠标操作都被忽略
	<code>QRect rubberBandRect()</code>	返回选择矩形框
	<code>void setRubberBandSelectionMode()</code>	选择模式，参数为枚举类型 <code>Qt::ItemSelectionMode</code>
	<code>QGraphicsItem* itemAt()</code>	获取视图坐标系中某个位置处的图形项
坐标映射	<code>QList&lt;QGraphicsItem*&gt; items()</code>	获取场景中的所有、或者某个选择区域内图形项的列表
	<code>QPoint mapFromScene()</code>	将场景中的一个坐标转换为视图的坐标
	<code>QPointF mapToScene()</code>	将视图中的一个坐标转换为场景的坐标

#### 2. QGraphicsScene 类的主要接口函数

`QGraphicsScene` 是用于管理图形项的场景，是图形项的容器，有添加、删除图形项的函数，管理图形项的各种函数。表 8-7 是 `QGraphicsScene` 的主要接口函数（仅列出函数的返回数据类型，

省略了输入参数)。

表 8-7 QGraphicsScene 主要函数功能说明

分组	函数	功能描述
场景	void setSceneRect()	设置场景的矩形区
分组	QGraphicsItemGroup* createItemGroup()	创建图形项组
	void destroyItemGroup()	解除一个图形项组
输入焦点	QGraphicsItem * focusItem()	返回当前获得焦点的图形项
	void clearFocus()	去除选择焦点
	bool hasFocus()	视图是否有焦点
图形项操作	void addItem()	添加一个已经创建的图形项
	void removeItem()	删除图形项
	void clear()	清除所有图形项
	QGraphicsItem* mouseGrabberItem()	返回鼠标抓取的图形项
	QList<QGraphicsItem *> selectedItems()	返回选择的图形项列表
	void clearSelection()	清除所有选择
	QGraphicsItem * itemAt()	获取某个位置处的顶层图形项
	QList<QGraphicsItem *> items()	返回某个矩形区域、多边形等选择区域内的图形项列表
添加图形项	QGraphicsEllipseItem * addEllipse()	添加一个椭圆
	QGraphicsLineItem * addLine()	添加一条直线
	QGraphicsPathItem * addPath()	添加一个绘图路径 (QPainterPath)
	QGraphicsPixmapItem * addPixmap()	添加一个图片
	QGraphicsPolygonItem * addPolygon()	添加一个多边形
	QGraphicsRectItem * addRect()	添加一个矩形
	QGraphicsSimpleTextItem * addSimpleText()	添加简单文字
	QGraphicsTextItem * addText()	添加字符串
	QGraphicsProxyWidget * addWidget()	添加界面组件

### 3. 图形项

QGraphicsItem 是所有图形项的基类，用户也可以从 QGraphicsItem 继承定义自己的图形项。Qt 定义了一些常见的图形项，这些常见的图形项的类的继承关系如图 8-18 所示。

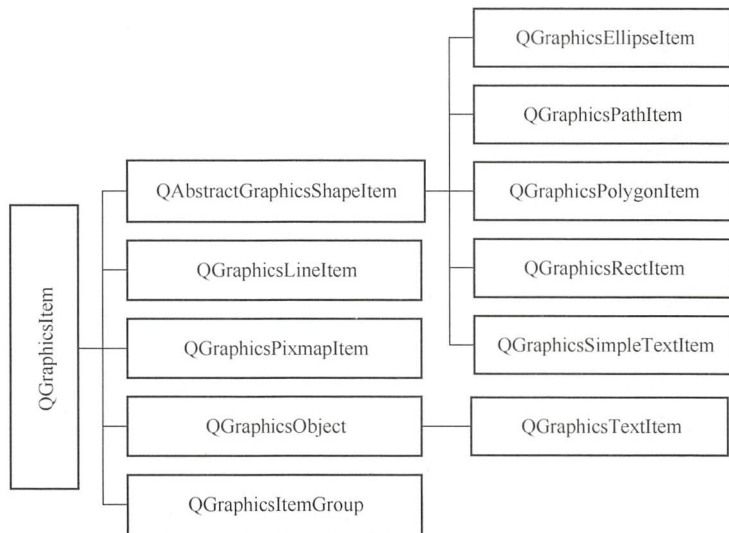


图 8-18 常见图形项类的继承关系



QGraphicsItem 类提供了图形项操作的函数，常见的函数见表 8-8（仅列出函数的返回数据类型，省略了输入参数）。

表 8-8 QGraphicsItem 主要函数功能说明

分组	函数	功能描述
属性设置	void setFlags()	设置图形项的操作属性，例如，可选择、可移动等
	void setOpacity()	设置透明度
	void setGraphicsEffect()	设置图形效果
	void setSelected()	图形项是否被选中
	void setData()	用户自定义数据
坐标	void setX()	图形项的 X 坐标
	void setY()	图形项的 Y 坐标
	void setZValue()	图形项的 Z 值，Z 值控制图形项的叠放次序
	void setPos()	图形项在父项中的位置
	QPointF scenePos()	返回图形项在场景中的坐标，相当于调用 mapToScene(0, 0)
坐标变换	void resetTransform()	复位坐标系，取消所有坐标变换
	void setRotation()	旋转一定角度，参数为正数时表示顺时针旋转
	void setScale()	按比例缩放，缺省值为 1
坐标映射	QPointF mapFromItem()	将另一个图形项的一个点映射到本图形项的坐标系
	QPointF mapFromParent()	将父项的一个点映射到本图形项的坐标系
	QPointF mapFromScene()	将场景中的一个点映射到本图形项的坐标系
	QPointF mapToItem()	将本图形项内的一个点映射到另一个图形项的坐标系
	QPointF mapToParent()	将本图形项内的一个点映射到父项坐标系
	QPointF mapToScene()	将本图形项内的一个点映射到场景坐标系

setFlags()函数可以设置一个图形项的操作标志，包括可选择、可移动、可获取焦点等，如：

```
item->setFlags(QGraphicsItem::ItemIsMovable
              | QGraphicsItem::ItemIsSelectable
              | QGraphicsItem::ItemIsFocusable);
```

setPos()函数设置图形项在父项中的坐标，如果没有父项，就是在场景中的坐标。

setZValue()控制图形项的叠放次序，当有多个图形项有重叠时，zValue 越大的，越显示在前面。

图形项可以通过 setRotation()进行旋转，通过 setScale()进行缩放。

图形项还可以与其他图形项、父项和场景之间进行坐标转换，这些将在实例程序中讲解。

8.3.4 Graphics View 程序基本结构和功能实现

1. 实例程序功能

创建一个实例 samp8\_4，是一个以 QMainWindow 为基类的窗口程序，其运行时界面如图 8-19 所示。

实例程序 samp8\_4 的主要功能包括以下几点。

- 工作区是一个从 QGraphicsView 继承的自定义类 QWGraphicsView，作为绘图的视图组件。
- 创建一个 QGraphicsScene 场景，场景的大小就是图中的实线矩形框的大小。
- 改变窗口大小，当视图大于场景时，矩形框总是居于图形视图的中央；当视图小于场景时，在视图窗口自动出现卷滚条。
- 蓝色椭圆正好处于场景的中间，红色圆形位于场景的右下角。当图形项位置不在场景的矩形框中时，图形项也是可以显示的。



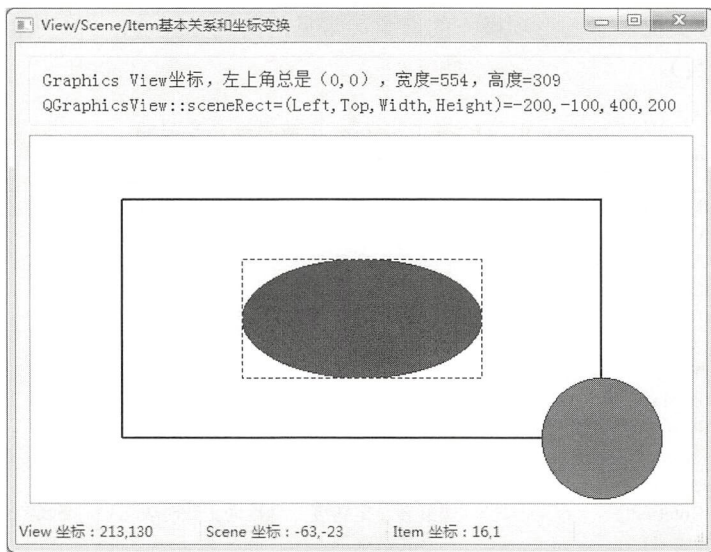


图 8-19 实例 samp8\_4 运行时界面

- 当鼠标在窗口上移动时，会在状态栏显示当前光标位置的视图坐标和场景坐标，在某个图形项上单击鼠标时，还会显示在图形项中的局部坐标。

这个实例演示了 Graphics View 绘图几个类的基本使用方法，演示视图、场景和绘图项 3 个坐标系的关系，以及它们之间的坐标转换。

## 2. 自定义图形视图组件

QGraphicsView 是 Qt 的图形视图组件，在 UI 设计器的 Display Widgets 分组里可以拖放一个 QGraphicsView 组件到窗口上。但是本实例中需要实现鼠标在 QGraphicsView 上移动时就显示当前光标的坐标，这涉及 mouseMoveEvent() 事件的处理。QGraphicsView 没有与 mouseMoveEvent() 相关的信号，因而无法定义槽函数与此事件相关联。

为此，从 QGraphicsView 继承定义一个类 QWGraphicsView，实现 mouseMoveEvent() 事件和 mousePressEvent() 事件，并把鼠标事件转换为信号，这样就可以在主程序里设计槽函数响应这些鼠标事件。下面是 QWGraphicsView 类的定义：

```
class QWGraphicsView : public QGraphicsView
{
    Q_OBJECT
protected:
    void mouseMoveEvent(QMouseEvent *event);
    void mousePressEvent(QMouseEvent *event);
public:
    QWGraphicsView(QWidget *parent = 0);
signals:
    void mouseMovePoint(QPoint point);
    void mouseClicked(QPoint point);
};
```

mouseMoveEvent() 是鼠标移动事件，其实现代码如下：

```
void QWGraphicsView::mouseMoveEvent(QMouseEvent *event)
{ // 鼠标移动事件
    QPoint point=event->pos(); //QGraphicsView 的坐标
    emit mouseMovePoint(point); //发射信号
    QGraphicsView::mouseMoveEvent(event);
}
```

在此事件响应代码里, 通过事件的 pos() 函数获取鼠标光标在视图中的坐标 point, 然后作为参数发射 mouseMovePoint(point) 信号。这样, 若在其他地方编写槽函数与此信号关联, 就可以对鼠标移动事件作出响应。

mousePressEvent() 是鼠标按键按下的事件, 其实现代码如下:

```
void QWGraphicsView::mousePressEvent(QMouseEvent *event)
{ // 鼠标左键按下事件
    if (event->button() == Qt::LeftButton)
    {
        QPoint point=event->pos(); //QGraphicsView 的坐标
        emit mouseClicked(point); //发射信号
    }
    QGraphicsView::mousePressEvent(event);
}
```

在此事件响应代码里, 首先判断是否是鼠标左键按下, 然后通过事件的 pos() 函数获取鼠标光标在视图中的坐标 point, 然后作为参数发射 mouseClicked(point) 信号。

### 3. 主窗口类定义与 QGraphicsView 组件升级

主窗口是一个从 QMainWindow 继承的类, 重定义了 resizeEvent() 事件, 对窗口改变大小的事件作出响应。槽函数 on\_mouseMovePoint() 响应鼠标在图形视图上移动的事件信号, 显示视图坐标和场景坐标; 槽函数 on\_mouseClicked() 响应鼠标单击信号, 显示图形项的局部坐标; iniGraphicsSystem() 用于创建 Graphics View 结构的各个对象。

MainWindow 类的完整定义如下:

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QGraphicsScene *scene;
    QLabel *labViewCord;
    QLabel *labSceneCord;
    QLabel *labItemCord;
    void iniGraphicsSystem(); //创建 Graphics View 的各项
protected:
    void resizeEvent(QResizeEvent *event);
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    void on_mouseMovePoint(QPoint point);
    void on_mouseClicked(QPoint point);
private:
    Ui::MainWindow *ui;
};
```

主窗口是采用 UI 设计器进行可视化设计的。在设计界面时，先从组件面板里放一个 QGraphicsView 组件到窗口上。但是我们要用的是从 QGraphicsView 继承的自定义图形视图组件 QWGraphicsView，需要将 QGraphicsView 升级为 QWGraphicsView。

在设计的窗口上选中放置的 QGraphicsView 组件，单击右键，在快捷菜单中选择“Promote to...”，出现如图 8-20 的对话框。在其中选择基类名称 QGraphicsView，升级类名称输入 QWGraphicsView，然后单击“Promote”按钮，就可以将窗口上的 QGraphicsView 组件升级为 QWGraphicsView 组件。这是使用自定义界面组件的一种方法，在第 12 章会详细介绍。

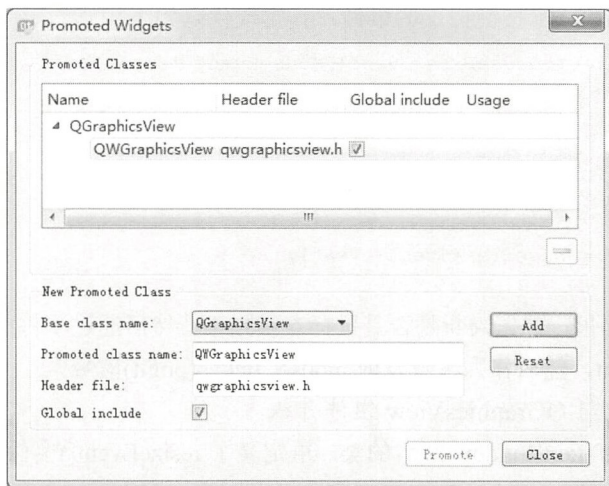


图 8-20 QGraphicsView 组件升级为自定义 QWGraphicsView 类

#### 4. 窗口初始化

下面是主窗口的构造函数的代码：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    labViewCord=new QLabel("View 坐标: ");
    labViewCord->setMinimumWidth(150);
    ui->statusBar->addWidget(labViewCord);
    labSceneCord=new QLabel("Scene 坐标: ");
    labSceneCord->setMinimumWidth(150);
    ui->statusBar->addWidget(labSceneCord);
    labItemCord=new QLabel("Item 坐标: ");
    labItemCord->setMinimumWidth(150);
    ui->statusBar->addWidget(labItemCord);
    ui->View->setCursor(Qt::CrossCursor);
    ui->View->setMouseTracking(true);
    ui->View->setDragMode(QGraphicsView::RubberBandDrag);
    QObject::connect(ui->View,SIGNAL(mouseMovePoint(QPoint)),
                    this, SLOT(on_mouseMovePoint(QPoint)));
    QObject::connect(ui->View,SIGNAL(mouseClicked(QPoint)),
                    this, SLOT(on_mouseClicked(QPoint)));
    iniGraphicsSystem();
}
```

槽函数 `on_mouseMovePoint()` 响应鼠标在图形视图上移动的 `mouseMovePoint()` 信号，实现代码如下：

```
void MainWindow::on_mouseMovePoint(QPoint point)
{ // 鼠标移动事件, point 是 GraphicsView 的坐标, 物理坐标
    labViewCord->setText(QString::asprintf("View 坐标: %d,%d", point.x(),point.y()));
    QPointF pointScene=ui->View->mapToScene(point); // 转换到 Scene 坐标
    labSceneCord->setText(QString::asprintf("Scene 坐标: %.0f,%.0f",
pointScene.x(),pointScene.y()));
}
```

信号传递的参数 `point` 就是鼠标在图形视图中的坐标，使用视图组件的 `mapToScene()` 函数可以将此坐标转换为场景中的坐标，这两个坐标在状态栏上显示。

槽函数 `on_mouseClicked()` 响应鼠标在图形视图上单击的 `mouseClicked()` 信号，代码如下：

```
void MainWindow::on_mouseClicked(QPoint point)
{ // 鼠标单击事件
    QPointF pointScene=ui->View->mapToScene(point); // 转换到 Scene 坐标
    QGraphicsItem *item=NULL;
    item=scene->itemAt(pointScene,ui->View->transform()); // 获取光标下的图形项
    if (item != NULL) // 有图形项
    {
        QPointF pointItem=item->mapFromScene(pointScene); // 图形项局部坐标
        labItemCord->setText(QString::asprintf("Item 坐标: %.0f,%.0f",
pointItem.x(),pointItem.y()));
    }
}
```

信号传递的参数 `point` 就是鼠标光标在图形视图中的坐标，先使用视图组件的 `mapToScene()` 函数将此坐标转换为场景中的坐标 `pointScene`，然后通过场景对象的 `itemAt()` 函数获得光标下的图形项。如果鼠标光标下有图形项，就用图形项的 `mapFromScene()` 函数将 `pointScene` 转换为图形项的局部坐标 `pointItem`，并在状态栏上显示。

另外还定义了窗口的 `resizeEvent()` 事件的响应函数，以便在窗口变化大小时，显示视图区域的大小，以及场景的大小信息。其代码如下：

```
void MainWindow::resizeEvent(QResizeEvent *event)
{ // 窗口变化大小时的事件
    ui->labViewSize->setText(QString::asprintf("Graphics View 坐标, 左上角总是 (0,0), 宽度=%d, 高度=%d",ui->View->width(),ui->View->height()));
    QRectF rectF=ui->View->sceneRect(); // Scene 的矩形区
    ui->labSceneRect->setText(QString::asprintf("QGraphicsView::sceneRect=
(Left,Top,Width,Height)=%.0f,%.0f,%.0f,%.0f",rectF.left(),rectF.top(),
rectF.width(),rectF.height()));
}
```

## 5. Graphics View 系统初始化

构造函数中调用的 `iniGraphicsSystem()` 用于创建 Graphics View 结构中的其他元素，包括场景和多个图形项。其代码如下：

```
void MainWindow::iniGraphicsSystem()
{ // 构造 Graphics View 的各项
    QRectF rect(-200,-100,400,200); // 左上角坐标, 宽度, 高度
```



```

    scene=new QGraphicsScene(rect); //scene 逻辑坐标系定义
    ui->View->setScene(scene);
//画一个矩形框, 大小等于 scene
    QGraphicsRectItem *item=new QGraphicsRectItem(rect);
    item->setFlags(QGraphicsItem::ItemIsSelectable //设置 flags
        | QGraphicsItem::ItemIsFocusable);

    QPen pen;
    pen.setWidth(2);
    item->setPen(pen);
    scene->addItem(item);
//画一个位于 scene 中心的椭圆, 测试局部坐标
    QGraphicsEllipseItem *item2=new QGraphicsEllipseItem(-100,-50,200,100);
    item2->setPos(0,0);
    item2->setBrush(QBrush(Qt::blue));
    item2->setFlags(QGraphicsItem::ItemIsMovable
        | QGraphicsItem::ItemIsSelectable
        | QGraphicsItem::ItemIsFocusable);
    scene->addItem(item2);
//画一个圆, 中心位于 scene 的边缘
    QGraphicsEllipseItem *item3=new QGraphicsEllipseItem(-50,-50,100,100);
    item3->setPos(rect.right(),rect.bottom());
    item3->setBrush(QBrush(Qt::red));
    item3->setFlags(QGraphicsItem::ItemIsMovable
        | QGraphicsItem::ItemIsSelectable
        | QGraphicsItem::ItemIsFocusable);
    scene->addItem(item3);
    scene->clearSelection();
}

```

可视化设计窗体时, 将 `QWGraphicsView` 类对象的名称命名为 `View`, 并且自动填充主窗口的工作区。创建场景并与 `View` 关联的代码如下:

```

QRectF rect(-200,-100,400,200); //左上角坐标, 宽度, 高度
scene=new QGraphicsScene(rect); //scene 逻辑坐标系定义
ui->View->setScene(scene);

```

这里用一个矩形定义了创建的场景的坐标系, 表示场景的左上角坐标是(-200, -100), 场景宽度为 400, 高度为 200, 这样, 场景的中心点是(0, 0), 这是场景的坐标系。

创建了一个矩形框图形项 `item`, 矩形框的大小就等于创建的场景的大小, 矩形框不能移动。

创建的第二个图形项 `item2` 是一个椭圆, 椭圆的左上角坐标是(-100, -50), 宽度 200, 高度 100, 所以椭圆的中心是(0, 0), 这是图形项的局部坐标系。再采用 `setPos(0, 0)` 设置椭圆在场景中的位置, 若不调用 `setPos()` 函数设置图形项在场景中的位置, 缺省为位置为(0, 0)。椭圆设置为可移动、可选择、可以获得焦点。

创建的第三个图形项 `item3` 是一个圆, 圆的左上角坐标是(-50, -50), 宽度 100, 高度 100, 所以圆的中心是(0, 0), 这是图形项的局部坐标系。再采用 `setPos()` 设置圆在场景中的位置。

```

item3->setPos(rect.right(),rect.bottom());

```

其中心位置是场景的右下角, 圆的一部分区域是超出了场景的矩形区域的, 但是整个圆还是可以正常显示的。

从这个实例程序可以看到 `Graphics View` 结构中场景与视图的关系, 如何创建图形项组件, 场

景、视图、图形项各自的坐标系以及坐标系之间的转换关系。

### 8.3.5 Graphics View 绘图程序实例

#### 1. 实例功能

实例 samp8\_4 只是演示了 Graphics View 的基本结构和 3 个坐标系的概念, 为了演示 Graphics View 结构编程的更多功能, 创建了实例程序 samp8\_5。

实例 samp8\_5 是一个基于 Graphics View 结构的简单绘图程序, 通过这个实例可以发现 Graphics View 图形编程更多功能的使用方法。程序运行界面如图 8-21 所示。

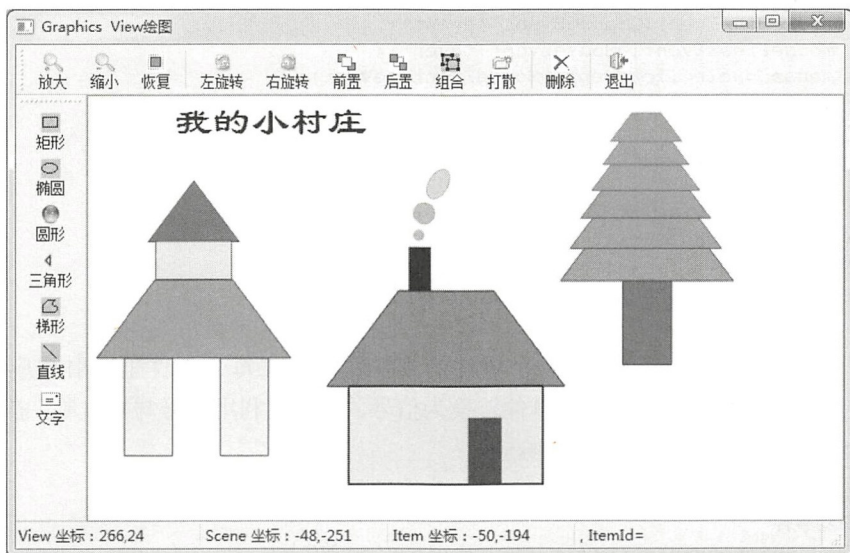


图 8-21 基于 Graphics View 结构的绘图程序 samp8\_5

这个实例程序具有如下的功能。

- 可以创建矩形、椭圆、圆形、三角形、梯形、直线、文字等基本图形项。
- 每个图形项都可以被选择和拖动。
- 图形项或整个整个视图可以缩放和旋转。
- 图形项重叠时, 可以调整前置或后置。
- 多个图形项可以组合, 也可以解除组合。
- 可以删除选择的图形项。
- 鼠标在视图上移动时, 会在状态栏显示视图坐标和场景坐标。
- 鼠标单击某个图形项时, 会显示图形项的局部坐标, 也会显示图形项的文字描述和编号。
- 双击某个图形项时, 会根据图形项的类型调用颜色对话框或字体对话框, 设置图形项的填充颜色、线条颜色或文字的字体。
- 选中某个图形项时, 可以进行按键操作, Delete 键删除图形项, PgUp 放大, PgDn 缩小, 空格键旋转 90°, 上下左右光标键移动图形项。

## 2. 自定义图形视图类 QWGraphicsView

与实例 samp8\_4 类似，从 QGraphicsView 类继承定义一个图形视图类 QWGraphicsView，在自定义类中添加鼠标和按键事件的处理，将鼠标和按键事件转换为信号，以便在主程序中设计槽函数做相应的处理。

下面是 QWGraphicsView 类的定义，处理了 mouseMoveEvent()、mousePressEvent()、mouseDoubleClickEvent()和 keyPressEvent()事件，定义了相应的信号，在事件处理里发射信号。

```
class QWGraphicsView : public QGraphicsView
{
    Q_OBJECT
protected:
    void mouseMoveEvent(QMouseEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseDoubleClickEvent(QMouseEvent *event);
    void keyPressEvent(QKeyEvent *event);
public:
    QWGraphicsView(QWidget *parent = 0);
signals:
    void mouseMovePoint(QPoint point); //鼠标移动
    void mouseClicked(QPoint point); //鼠标单击
    void mouseDoubleClick(QPoint point); //双击事件
    void keyPress(QKeyEvent *event); //按键事件
};
```

下面是 QWGraphicsView 类的 4 个事件的实现代码，在每个事件里发射相应的信号。在 QWGraphicsView 类里，将鼠标和键盘事件转换为信号，就可以利用信号与槽机制，在主程序里设计槽函数对相应的鼠标和键盘操作进行响应。

```
void QWGraphicsView::mouseMoveEvent(QMouseEvent *event)
{ //鼠标移动事件
    QPoint point=event->pos(); //QGraphicsView 的坐标
    emit mouseMovePoint(point); //发射信号
    QGraphicsView::mouseMoveEvent(event);
}

void QWGraphicsView::mousePressEvent(QMouseEvent *event)
{ //鼠标按键按下事件
    if (event->button()==Qt::LeftButton)
    {
        QPoint point=event->pos(); //QGraphicsView 的坐标
        emit mouseClicked(point); //发射信号
    }
    QGraphicsView::mousePressEvent(event);
}

void QWGraphicsView::mouseDoubleClickEvent(QMouseEvent *event)
{ //鼠标双击事件
    if (event->button()==Qt::LeftButton)
    {
        QPoint point=event->pos(); //QGraphicsView 的坐标
        emit mouseDoubleClick(point); //发射信号
    }
    QGraphicsView::mouseDoubleClickEvent(event);
}
```

```
void QWGraphicsView::keyPressEvent(QKeyEvent *event)
{ //按键事件
    emit keyPress(event); //发射信号
    QGraphicsView::keyPressEvent(event);
}
```

### 3. 主窗口设计

主窗口是一个从 QMainWindow 继承的窗口类 MainWindow，采用可视化设计。设计的 Action 如图 8-22 所示，利用这些 Action 创建两个工具栏，一个用于图形项的创建，一个用于图形项的各种操作。主窗口工作区的 QWGraphicsView 组件是从 QGraphicsView 组件升级而来的。

Name	Used	Text	Shortcut	Checkable	ToolTip
actItem_Rect	<input checked="" type="checkbox"/>	矩形		<input type="checkbox"/>	添加矩形
actItem_Ellipse	<input checked="" type="checkbox"/>	椭圆		<input type="checkbox"/>	添加椭圆型
actItem_Line	<input checked="" type="checkbox"/>	直线		<input type="checkbox"/>	添加直线
actEdit_Delete	<input checked="" type="checkbox"/>	删除		<input type="checkbox"/>	删除选中的图元
actQuit	<input checked="" type="checkbox"/>	退出		<input type="checkbox"/>	退出本系统
actItem_Pixmap	<input type="checkbox"/>	图片		<input type="checkbox"/>	添加图片
actItem_Text	<input checked="" type="checkbox"/>	文字		<input type="checkbox"/>	添加文字
actEdit_Front	<input checked="" type="checkbox"/>	前置		<input type="checkbox"/>	居于最前面
actEdit_Back	<input checked="" type="checkbox"/>	后置		<input type="checkbox"/>	居于最后面
actItem_Polygon	<input checked="" type="checkbox"/>	梯形		<input type="checkbox"/>	添加梯形
actZoomIn	<input checked="" type="checkbox"/>	放大		<input type="checkbox"/>	放大
actZoomOut	<input checked="" type="checkbox"/>	缩小		<input type="checkbox"/>	缩小
actRotateLeft	<input checked="" type="checkbox"/>	左旋转		<input type="checkbox"/>	左旋转
actRotateRight	<input checked="" type="checkbox"/>	右旋转		<input type="checkbox"/>	右旋转
actRestore	<input checked="" type="checkbox"/>	恢复		<input type="checkbox"/>	恢复大小
actGroup	<input checked="" type="checkbox"/>	组合		<input type="checkbox"/>	组合
actGroupBreak	<input checked="" type="checkbox"/>	打散		<input type="checkbox"/>	取消组合
actItem_Circle	<input checked="" type="checkbox"/>	圆形		<input type="checkbox"/>	圆形
actItem_Triangle	<input checked="" type="checkbox"/>	三角形		<input type="checkbox"/>	三角形

图 8-22 主窗口的 Action

主窗口类 MainWindow 的定义的主要部分如下（省略了 Action 的槽函数的定义）：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    static const int ItemId = 1; //图形项自定义数据的 key
    static const int ItemDescription = 2; //图形项自定义数据的 key
    int seqNum=0; //用于图形项的编号，每个图形项有一个编号
    int frontZ=0; //用于 bring to front
    int backZ=0; //用于 bring to back
    QGraphicsScene *scene;
    QLabel *labViewCord;
    QLabel *labSceneCord;
    QLabel *labItemCord;
    QLabel *labItemInfo;
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    void on_mouseMovePoint(QPoint point); //鼠标移动
    void on_mouseClicked(QPoint point); //鼠标单击
    void on_mouseDoubleClick(QPoint point); //鼠标双击
    void on_keyPress(QKeyEvent *event); //按键
private:
```



```
Ui::MainWindow *ui;
};
```

ItemId 和 ItemDescription 是用于定义图形项的自定义数据的键。

变量 seqNum 用于给每个图形项编号，每个图形项有一个唯一编号。

变量 frontZ 用于设置图形项的叠放顺序，数值越大，越在前面显示。

变量 backZ 用于设置图形项的叠放顺序，数值越小，越在后面显示。

定义了 4 个私有槽函数，用于响应 QGraphicsView 的 4 个信号，实现鼠标移动、单击、双击、按键时的处理。

- on\_mouseMovePoint()槽函数响应绘图视图的 mouseMovePoint()信号，在鼠标移动时显示鼠标光标处的视图坐标和场景坐标。
- on\_mouseClicked()槽函数响应绘图视图的 mouseClicked()信号，在鼠标单击时，显示图形项的局部坐标和图形项存储的自定义信息，如图形项编号和描述信息。
- on\_mouseDoubleClick()槽函数响应绘图视图的 mouseDoubleClick()信号，在某个图形项上双击时，根据图形项的类型调用颜色对话框或字体对话框，设置填充颜色、线条颜色或字体。
- on\_keyPress()槽函数响应绘图视图的 keyPress()信号，在选中某个图形项时，用按键实现缩放、删除、移动等操作。

MainWindow 的构造函数代码如下：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setUpUi(this);
    labViewCord=new QLabel("View 坐标: "); //创建状态栏标签
    labViewCord->setMinimumWidth(150);
    ui->statusBar->addWidget(labViewCord);
    labSceneCord=new QLabel("Scene 坐标: ");
    labSceneCord->setMinimumWidth(150);
    ui->statusBar->addWidget(labSceneCord);
    labItemCord=new QLabel("Item 坐标: ");
    labItemCord->setMinimumWidth(150);
    ui->statusBar->addWidget(labItemCord);
    labItemInfo=new QLabel("ItemInfo: ");
    labItemInfo->setMinimumWidth(200);
    ui->statusBar->addWidget(labItemInfo);

    scene=new QGraphicsScene(-300,-200,600,200); //创建 QGraphicsScene
    ui->View->setScene(scene); //与 view 关联
    ui->View->setCursor(Qt::CrossCursor); //设置鼠标光标形状
    ui->View->setMouseTracking(true);
    ui->View->setDragMode(QGraphicsView::RubberBandDrag);
    this->setCentralWidget(ui->View);

    QObject::connect(ui->View, SIGNAL(mouseMovePoint(QPoint)),
                     this, SLOT(on_mouseMovePoint(QPoint)));
    QObject::connect(ui->View, SIGNAL(mouseClicked(QPoint)),
                     this, SLOT(on_mouseClicked(QPoint)));
    QObject::connect(ui->View, SIGNAL(mouseDoubleClick(QPoint)),
                     this, SLOT(on_mouseDoubleClick(QPoint)));
    QObject::connect(ui->View, SIGNAL(keyPress(QKeyEvent*)),
```

```

        this, SLOT(on_keyPress(QKeyEvent*))));
    qsrand(Qtime::currentTime().second()); //随机数初始化
}

```

在构造函数里，首先是创建状态栏上的标签，然后创建场景并与视图关联，再将视图组件的 4 个信号与 4 个槽函数关联。

qsrand()用于随机数初始化，这里用当前时间的秒作为随机数种子。

#### 4. 图形项的创建

主窗口左侧工具栏上的按钮用于创建各种标准的图形项。创建矩形使用 QGraphicsRectItem，创建椭圆和圆使用 QGraphicsEllipseItem，创建三角形和梯形使用 QGraphicsPolygonItem，创建直线使用 QGraphicsLineItem，创建文字使用 QGraphicsTextItem。还有一些其他的标准图形项类，程序里未全部涉及。

创建各图形项的代码基本类似，以创建椭圆的代码为例进行说明。

```

void MainWindow::on_actItem_Ellipse_triggered()
{ //添加一个椭圆
    QGraphicsEllipseItem *item=new QGraphicsEllipseItem(-50,-30,100,60);
    item->setFlags(QGraphicsItem::ItemIsMovable
                | QGraphicsItem::ItemIsSelectable
                | QGraphicsItem::ItemIsFocusable);
    item->setBrush(QBrush(Qt::blue)); //填充颜色
    item->setZValue(++frontZ); //用于叠放顺序
    item->setPos(-50+(qrand() % 100),-50+(qrand() % 100)); //初始位置
    item->setData(ItemId,++seqNum); //自定义数据, ItemId 键
    item->setData(ItemDescription,"椭圆"); //自定义数据, ItemDescription 键

    scene->addItem(item);
    scene->clearSelection();
    item->setSelected(true);
}

```

椭圆图形类是 QGraphicsEllipseItem。创建一个椭圆图形实例 item，用 setFlags()函数设置为可移动、可选择和可获取焦点；setBrush()用于设置填充色；setZValue()用于设置 ZValue，这个参数控制叠放顺序，当有多个图形项叠加在一起时，ZValue 值最大的显示在最前面；setPos()函数设置图形项在场景中的位置，这里使用了随机数函数 qrand()。

setData()函数用于设置图形项的自定义参数，setData()函数原型为：

```
void QGraphicsItem::setData(int key, const QVariant &value)
```

key 是一个整数，value 是 QVariant 类型，key 和 value 是一个键值对。所以，使用 setData()一次可以设置一个键值对，可以为一个图形项设置多个自定义键值对。程序里设置了两个自定义数据：

```

item->setData(ItemId,++seqNum); //自定义数据, ItemId 键
item->setData(ItemDescription,"椭圆"); //自定义数据, ItemDescription 键

```

ItemId 是图形项的编号，ItemDescription 是图形项的描述，两个都是在 MainWindow 类中定义的静态变量。这样，每个图形项有一个唯一的编号，有一个文字描述。在窗口上单击某个图形项时，会提取这两个自定义数据在状态栏上显示。

工具栏上的按钮还可以创建其他一些图形项，代码与此基本类似，这里不再详细介绍。

### 5. 图形项操作

主窗口的另一个工具栏上的按钮实现图形项的缩放、旋转、组合等操作。

#### • 缩放

图形项的缩放使用 `QGraphicsItem` 的 `setScale()` 函数，参数大于 1 是放大，小于 1 是缩小。例如，下面是实现放大的程序。

```
void MainWindow::on_actZoomIn_triggered()
{ //放大
    int cnt=scene->selectedItems().count();
    if (cnt==1)
    {
        QGraphicsItem* item=scene->selectedItems().at(0);
        item->setScale(0.1+item->scale());
    }
    else
        ui->View->scale(1.1,1.1);
}
```

`QGraphicsScene::selectedItems()` 返回场景中选中的图形项的列表，如果只有一个图形项被选中，就用 `QGraphicsItem` 的 `setScale()` 对图形项进行放大，在原来的缩放系数 `scale()` 基础上加 0.1 作为放大系数。如果选中的图形项个数大于 1 个，或没有图形项被选中，就用 `QGraphicsView` 的 `scale()` 函数对绘图视图进行放大。

#### • 旋转

图形项的旋转使用 `QGraphicsItem::setRotation()` 函数，参数为角度值，正值表示顺时针旋转，负值表示逆时针旋转。例如，下面是逆时针旋转的程序。

```
void MainWindow::on_actRotateLeft_triggered()
{ //逆时针旋转
    int cnt=scene->selectedItems().count();
    if (cnt==1)
    {
        QGraphicsItem* item=scene->selectedItems().at(0);
        item->setRotation(-30+item->rotation());
    }
    else
        ui->View->rotate(-30);
}
```

如果有一个图形项被选中，就对图形项进行旋转，否则对绘图视图进行旋转。

#### • 恢复坐标变换

缩放和旋转都是坐标变换，要取消所有变换恢复初始状态，调用 `QGraphicsItem` 或 `QGraphicsView` 的 `resetTransform()` 函数。下面是“恢复”按钮的响应代码。

```
void MainWindow::on_actRestore_triggered()
{ //取消所有变换
    int cnt=scene->selectedItems().count();
    if (cnt==1)
    {
        QGraphicsItem* item=scene->selectedItems().at(0);
```

```

        item->resetTransform();
    }
    else
        ui->View->resetTransform();
}

```

### • 叠放顺序

QGraphicsItem 的 `zValue()` 表示图形项在 Z 轴的值，若有多个图形项叠加在一起，`zValue()` 值最大的显示在最前面，`zValue()` 值最小的显示在最后面。用 `setZValue()` 函数可以设置这个属性值。下面是工具栏上的“前置”和“后置”按钮的代码。

```

void MainWindow::on_actEdit_Front_triggered()
{ //bring to front, 前置
    int cnt=scene->selectedItems().count();
    if (cnt>0)
    { //只处理选中的第 1 个图形项
        QGraphicsItem* item=scene->selectedItems().at(0);
        item->setZValue(++frontZ);
    }
}

void MainWindow::on_actEdit_Back_triggered()
{ //bring to back, 后置
    int cnt=scene->selectedItems().count();
    if (cnt>0)
    { //只处理选中的第 1 个图形项
        QGraphicsItem* item=scene->selectedItems().at(0);
        item->setZValue(--backZ);
    }
}

```

`frontZ` 和 `backZ` 是在 `MainWindow` 类中定义的私有变量，专门用于存储叠放次序的编号。`frontZ` 只增加，所以每增加一次都是最大值，设置该值的图形项就可以显示在最前面；`backZ` 只减少，所以每减小一次都是最小值，设置该值的图形项就可以显示在最后面。

### • 图形项的组合

可以将多个图形项组合为一个图形项，当作一个整体进行操作，如同 PowerPoint 软件里图形组合功能一样。使用 `QGraphicsItemGroup` 类实现多个图形项的组合，`QGraphicsItemGroup` 是 `QGraphicsItem` 的子类，所以实质上也是一个图形项。下面是工具栏上的“组合”按钮的响应代码：

```

void MainWindow::on_actGroup_triggered()
{ //组合
    int cnt=scene->selectedItems().count();
    if (cnt>1)
    {
        QGraphicsItemGroup* group =new QGraphicsItemGroup; //创建组合
        scene->addItem(group); //组合添加到场景中
        for (int i=0;i<cnt;i++)
        {
            QGraphicsItem* item=scene->selectedItems().at(0);
            item->setSelected(false); //清除选择虚线框
            item->clearFocus();
            group->addToGroup(item); //添加到组合
        }
    }
}

```



```

    }
    group->setFlags(QGraphicsItem::ItemIsMovable
                  | QGraphicsItem::ItemIsSelectable
                  | QGraphicsItem::ItemIsFocusable);
    group->setZValue(++frontZ);
    scene->clearSelection();
    group->setSelected(true);
}
}

```

当有多个图形项被选择时，创建一个 `QGraphicsItemGroup` 类型的实例 `group`，并添加到场景中，然后将选中的图形项逐一添加到 `group` 中。这样创建的 `group` 就是场景中的一个图形项，可以对其进行缩放、旋转等操作。

一个组合对象也可以被打散，使用 `QGraphicsScene` 的 `destroyItemGroup()` 函数可以打散一个组合对象，这个函数打散组合，删除组合对象，但是不删除原来组合里的图形项。下面是工具栏上的“打散”按钮的响应代码：

```

void MainWindow::on_actGroupBreak_triggered()
{ //break group, 打散组合
    int cnt=scene->selectedItems().count();
    if (cnt==1)
    {
        QGraphicsItemGroup *group;
        group=(QGraphicsItemGroup*) scene->selectedItems().at(0);
        scene->destroyItemGroup(group);
    }
}

```

这里假设在单击“打散”按钮时，选中的是一个组合对象，并没有做类型判断。

#### ● 图形项的删除

使用 `QGraphicsScene` 的 `removeItem()` 函数删除某个图形项，下面是工具栏上的“删除”按钮的响应代码。

```

void MainWindow::on_actEdit_Delete_triggered()
{ //删除所有选中的图形项
    int cnt=scene->selectedItems().count();
    if (cnt>0)
        for (int i=0;i<cnt;i++)
        {
            QGraphicsItem* item=scene->selectedItems().at(0);
            scene->removeItem(item); //删除图形项
        }
}

```

## 6. 鼠标与键盘操作

`MainWindow` 定义了 4 个槽函数与 `QWGraphicsView` 定义的 4 个信号进行关联，实现对鼠标移动、单击、双击和按键的响应。

#### ● 鼠标移动

鼠标在绘图视图上移动时，在状态栏显示光标处的视图坐标和场景坐标，`on_mouseMovePoint()` 槽函数的代码如下：

```
void MainWindow::on_mouseMovePoint(QPoint point)
{ //鼠标移动事件, point 是 QGraphicsView 的坐标, 物理坐标
    labViewCord->setText(QString::asprintf("View 坐标: %d,%d", point.x(),point.y()));
    QPointF pointScene=ui->View->mapToScene(point); //转换到 Scene 坐标
    labSceneCord->setText(QString::asprintf("Scene 坐标: %.0f,%.0f", pointScene.x(),
pointScene.y()));
}
```

参数 point 是鼠标光标在视图上的坐标, 用 QGraphicsView::mapToScene() 函数可以将此坐标转换为场景中的坐标。

#### • 鼠标单击

在视图上单击鼠标时, 如果选中一个图形项, 就会显示图形项的局部坐标, 并提取其自定义信息进行显示, on\_mouseClicked() 槽函数代码如下:

```
void MainWindow::on_mouseClicked(QPoint point)
{ //鼠标单击事件
    QPointF pointScene=ui->View->mapToScene(point); //转换到 Scene 坐标
    QGraphicsItem *item=NULL;
    item=scene->itemAt(pointScene,ui->View->transform()); //光标下的图形项
    if (item != NULL) //有图形项
    {
        QPointF pointItem=item->mapFromScene(pointScene); //图形项局部坐标
        labItemCord->setText(QString::asprintf("Item 坐标: %.0f,%.0f", pointItem.x(),
pointItem.y()));
        labItemInfo->setText(item->data(ItemDescription).toString()
            +", ItemId="+ item->data(ItemId).toString());
    }
}
```

首先将视图的坐标 point 转换为场景中的坐标 pointScene, 再利用 QGraphicsScene 的 itemAt() 函数获得光标处的图形项。利用 QGraphicsItem 的 mapFromScene() 函数将 pointScene 转换为图形项的局部坐标 pointItem。

在创建图形项时, 使用 setData() 函数设置了自定义数据的键值对, 这里用 data() 函数提取图形项的自定义的数据 ItemId 和 ItemDescription, 并进行显示。

#### • 鼠标双击

当鼠标双击某个图形项时, 希望根据图形项的类型, 调用不同的对话框进行图形项的设置。例如, 当图形项是矩形、圆形、梯形等有填充色的对象时, 打开一个颜色选择对话框, 设置其填充颜色; 当图形项是直线时, 设置其线条颜色; 当图形项是文字时, 打开一个字体对话框, 设置其字体。下面是 on\_mouseDoubleClick() 槽函数的代码。

```
void MainWindow::on_mouseDoubleClick(QPoint point)
{ //鼠标双击, 调用相应的对话框, 设置填充颜色、线条颜色或字体
    QPointF pointScene=ui->View->mapToScene(point); //转换到 Scene 坐标
    QGraphicsItem *item=NULL;
    item=scene->itemAt(pointScene,ui->View->transform()); //光标下的图形项
    if (item == NULL) //没有图形项
        return;

    switch (item->type()) //图形项的类型
    {
```

```

case QGraphicsRectItem::Type: //矩形框
{ //强制类型转换
    QGraphicsRectItem *theItem;
    theItem = qgraphicsitem_cast<QGraphicsRectItem*>(item);
    setBrushColor(theItem);
    break;
}
case QGraphicsEllipseItem::Type: //椭圆和圆都是此类型
{
    QGraphicsEllipseItem *theItem;
    theItem = qgraphicsitem_cast<QGraphicsEllipseItem*>(item);
    setBrushColor(theItem);
    break;
}
case QGraphicsPolygonItem::Type: //梯形和三角形
{
    QGraphicsPolygonItem *theItem;
    theItem = qgraphicsitem_cast<QGraphicsPolygonItem*>(item);
    setBrushColor(theItem);
    break;
}
case QGraphicsLineItem::Type: //直线, 设置线条颜色
{
    QGraphicsLineItem *theItem;
    theItem = qgraphicsitem_cast<QGraphicsLineItem*>(item);
    QPen pen=theItem->pen();
    QColor color=theItem->pen().color();
    color=QColorDialog::getColor(color,this,"选择线条颜色");
    if (color.isValid())
    {
        pen.setColor(color);
        theItem->setPen(pen);
    }
    break;
}
case QGraphicsTextItem::Type: //文字, 设置字体
{
    QGraphicsTextItem *theItem;
    theItem = qgraphicsitem_cast<QGraphicsTextItem*>(item);
    QFont font=theItem->font();
    bool ok=false;
    font=QFontDialog::getFont(&ok,font,this,"设置字体");
    if (ok)
        theItem->setFont(font);
    break;
}
}
}

```

双击鼠标时, 获取光标下的图形项 `item`, 由于不知道具体是什么类型的图形项, `item` 定义为 `QGraphicsItem`, 即所有图形项的父类。

`QGraphicsItem` 的函数 `type()` 返回图形项的类型, 每个具体的图形项类都需要重定义此函数, 例如自定义一个图形项时, 需要定义枚举常量 `Type` 和函数 `type`, 枚举常量 `Type` 的值必须大于 `UserType`, 示例代码如下:

```
class CustomItem : public QGraphicsItem
{
public:
    enum { Type = UserType + 1 };
    int type() const
    {
        // 使得 qgraphicsitem_cast 可以使用这个图形项
        return Type;
    }
    ...
};
```

根据 `item->type()` 值判断图形项的类型，若该值等于 `QGraphicsLineItem::Type`，说明这个 `item` 是一个 `QGraphicsLineItem` 类型实例，可以设置其线条颜色。

设置线条颜色可以调用 `QGraphicsLineItem::setPen()` 函数，但是 `QGraphicsItem` 没有 `setPen()` 函数。所以，需要使用图形项的强制类型转换函数 `qgraphicsitem_cast()` 将 `item` 转换为 `QGraphicsLineItem` 类型的 `theItem`，即：

```
QGraphicsLineItem *theItem;
theItem = qgraphicsitem_cast<QGraphicsLineItem*>(item);
```

然后就调用 `QColorDialog::getColor()` 函数选择颜色，设置为 `theItem` 的线条颜色。

`QGraphicsItem` 类同样没有 `setFont()` 和 `setBrush()` 函数，当选择的图形项是文字对象 `QGraphicsTextItem` 时，需要将其强制转换为 `QGraphicsTextItem` 类型的变量，调用字体选择对话框后，用 `QGraphicsTextItem::setFont()` 函数设置字体。

对于 `QGraphicsRectItem`、`QGraphicsEllipseItem` 或 `QGraphicsPolygonItem`，可以调用这 3 个类的 `setBrush()` 函数设置填充颜色，这里用一个函数 `setBrushColor()` 为 3 种不同类的对象进行填充色的设置。`setBrushColor()` 并不是使用不同类型参数的同名重载函数，而是一个在 `mainwindow.cpp` 文件中定义的一个独立的模板函数。

```
template<class T> void setBrushColor(T *item)
{ // 函数模板
    QColor color=item->brush().color();
    color=QColorDialog::getColor(color, NULL, "选择填充颜色");
    if (color.isValid())
        item->setBrush(QBrush(color));
}
```

编译器会自动根据调用 `setBrushColor()` 的参数的类型生成 3 个不同参数类型的函数，减少了代码的冗余性。

#### ● 按键操作

在选中一个图形项之后，可以通过键盘按键实现一些快捷操作，例如缩放、旋转、移动等。

`on_keyPress()` 槽函数实现对按键的响应，其代码如下：

```
void MainWindow::on_keyPress(QKeyEvent *event)
{ // 按键事件
    if (scene->selectedItems().count() != 1)
        return; // 没有选中的图形项，或选中的多于 1 个

    QGraphicsItem *item=scene->selectedItems().at(0);
```



```
if (event->key()==Qt::Key_Delete)//删除
    scene->removeItem(item);
else if (event->key()==Qt::Key_Space) //顺时针旋转 90 度
    item->setRotation(90+item->rotation());
else if (event->key()==Qt::Key_PageUp)//放大
    item->setScale(0.1+item->scale());
else if (event->key()==Qt::Key_PageDown) //缩小
    item->setScale(-0.1+item->scale());
else if (event->key()==Qt::Key_Left) //左移
    item->setX(-1+item->x());
else if (event->key()==Qt::Key_Right) //右移
    item->setX(1+item->x());
else if (event->key()==Qt::Key_Up) //上移
    item->setY(-1+item->y());
else if (event->key()==Qt::Key_Down) //下移
    item->setY(1+item->y());
}
```

这段代码限定只有一个图形项被选中时才可以执行键盘操作。

# Qt Charts

Qt Charts 是 Qt 提供的图表模块，在 Qt 5.7 以前只有商业版才有 Qt Charts，但是从 Qt 5.7 开始，社区版本也包含了 Qt Charts。Qt Charts 可以很方便地绘制常见的折线图、柱状图、饼图等图表，不用自己耗费时间和精力开发绘图组件或使用第三方组件了。

本章首先介绍 Qt Charts 的基本特点和功能，以画折线图为例详细说明 Qt Charts 各主要部件的操作方法，再介绍各种常用图表的绘图方法，最后介绍鼠标操作图形缩放等功能的实现。

## 9.1 Qt Charts 概述

### 9.1.1 Qt Charts 模块

Qt Charts 模块是一组易于使用的图表组件，它基于 Qt 的 Graphics View 架构，其核心组件是 QChartView 和 QChart。

QChartView 的父类是 QGraphicsView，就是 Graphics View 架构中的视图组件，所以，QChartView 是用于显示图表的视图。

QChart 的继承关系如图 9-1 所示，可以看到，QChart 是从 QGraphicsItem 继承而来的，所以，QChart 是一种图形项。

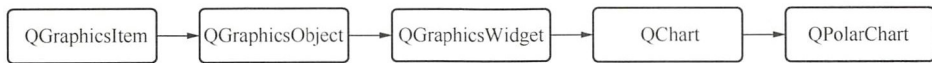


图 9-1 QChart 的继承关系

QPolarChart 是用于绘制极坐标图的图表类，它从 QChart 继承而来。

要在项目中使用 Qt Charts 模块，必须在项目的配置文件（.pro 文件）中增加下面的一行语句：

```
Qt += charts
```

在需要使用 QtCharts 的类的头文件或源程序文件中，要使用如下的包含语句：

```
#include <QtCharts>
using namespace QtCharts;
```

也可以使用宏定义：

```
#include <QtCharts>
Qt_CHARTS_USE_NAMESPACE
```

## 9.1.2 一个简单的 QChart 绘图程序

先用一个简单实例程序说明 QChart 绘图的基本原理。创建一个基于 QMainWindow 的应用程序 samp9\_1，主窗口上不放置任何组件。在主窗口类中只定义一个 createChart() 函数，在主窗口的构造函数中调用此函数，即：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    createChart();
}
```

createChart() 函数用于创建图表，其代码如下：

```
void MainWindow::createChart()
{ //创建图表
    QChartView *chartView=new QChartView(this); //创建 ChartView
    QChart *chart = new QChart(); //创建 Chart
    chart->setTitle("简单函数曲线");
    chartView->setChart(chart); //Chart 添加到 ChartView
    this->setCentralWidget(chartView);
    //创建折线序列
    QLineSeries *series0 = new QLineSeries();
    QLineSeries *series1 = new QLineSeries();
    series0->setName("Sin 曲线");
    series1->setName("Cos 曲线");
    chart->addSeries(series0); //序列添加到图表
    chart->addSeries(series1);
    //序列添加数值
    qreal t=0,y1,y2,intv=0.1;
    int cnt=100;
    for(int i=0;i<cnt;i++)
    {
        y1=qSin(t);//+qrand();
        series0->append(t,y1);
        y2=qSin(t+20);
        series1->append(t,y2);
        t+=intv;
    }
    //创建坐标轴
    QValueAxis *axisX = new QValueAxis; //X 轴
    axisX->setRange(0, 10); //设置坐标轴范围
    axisX->setTitleText("time(secs)"); //标题

    QValueAxis *axisY = new QValueAxis; //Y 轴
    axisY->setRange(-2, 2);
    axisY->setTitleText("value");

    chart->setAxisX(axisX, series0); //为序列设置坐标轴
    chart->setAxisY(axisY, series0);
    chart->setAxisX(axisX, series1); //为序列设置坐标轴
    chart->setAxisY(axisY, series1);
}
```

程序运行后界面如图 9-2 所示。

在 `createChart()` 函数里, 首先创建一个 `QChartView` 对象 `chartView`, 再创建一个 `QChart` 对象 `chart`, 将 `chart` 在 `chartView` 里显示, 使用下面一行语句:

```
chartView->setChart(chart);
```

图表上用于显示数据的称为序列 (series), 这里使用折线序列 `QLineSeries`, 创建了两个 `QLineSeries` 类型的序列, 并且将序列添加到 `chart` 中。

```
chart->addSeries(series0);
chart->addSeries(series1);
```

序列存储用于显示的数据, 所以需要为直线序列添加平面数据点的坐标数据。程序将生成正弦和余弦函数的数据作为序列的数据。

序列还需要坐标轴, 创建 `QValueAxis` 类型的坐标轴作为图表的  $X$  轴和  $Y$  轴, 调用 `QChart` 的 `setAxisX()` 和 `setAxisY()` 函数为两个序列分别设置  $X$  轴和  $Y$  轴。

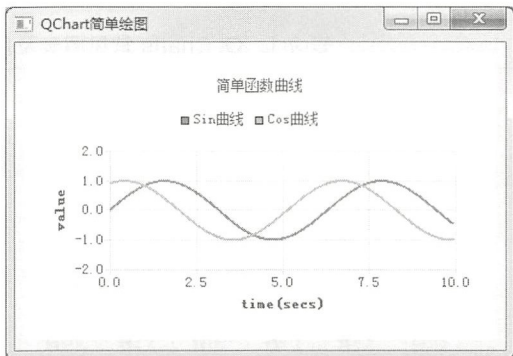


图 9-2 实例 samp9\_1 运行时界面

### 9.1.3 图表的主要组成部分

观察 `createChart()` 函数的内容和程序运行后的界面 (图 9-2), 可知 `QChartView` 是 `QChart` 的视图组件, 而一个 `QChart` 一般包括序列、坐标轴、图例、图表标题等部分。

#### 1. `QChartView` 的功能

`QChartView` 是 `QChart` 的视图组件, 类似于 `Graphics View` 架构中的 `QGraphicsView`。实际上, 在窗口设计界面上使用 `QChartView` 时, 就是先放置一个 `QGraphicsView` 组件, 然后升级为 `QChartView`。

`QChartView` 类定义的函数很少, 只有以下几个。

- `void setChart(QChart *chart)`, 设置一个 `QChart` 对象作为显示的图表。
- `QChart * chart()`, 返回 `QChartView` 当前设置的 `QChart` 类对象。
- `void setRubberBand(RubberBands &rubberBand)`, 设置选择框的类型, 即鼠标在视图组件上拖动选择范围的方式, 是一个 `QChartView::RubberBand` 枚举类型的组合, `QChartView::RubberBand` 枚举类型有以下几种取值:
  - `QChartView::NoRubberBand`——无选择框;
  - `QChartView::VerticalRubberBand`——垂向选择;
  - `QChartView::HorizontalRubberBand`——水平选择;
  - `QChartView::RectangleRubberBand`——矩形框选择;
- `RubberBands rubberBand()`, 返回设置的选择框类型。

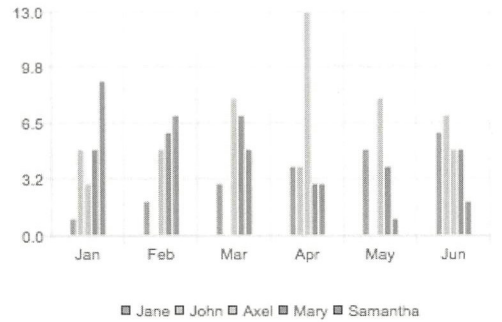
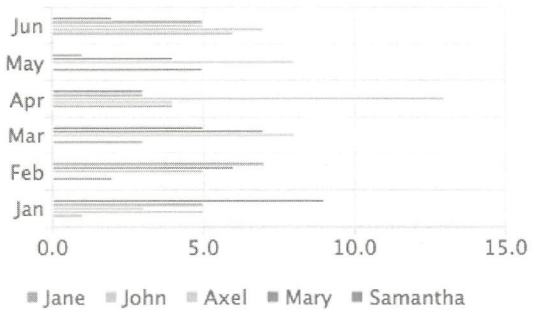
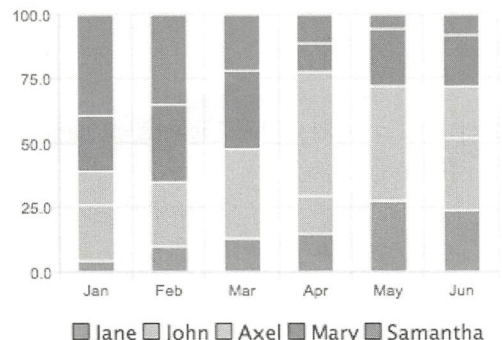
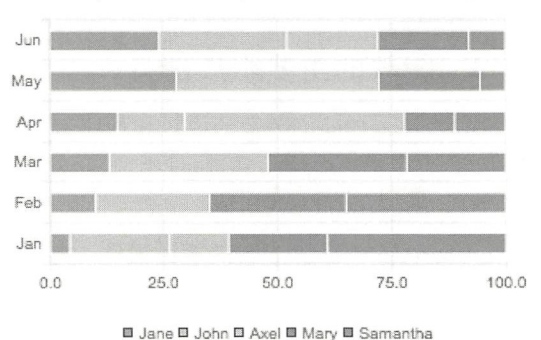
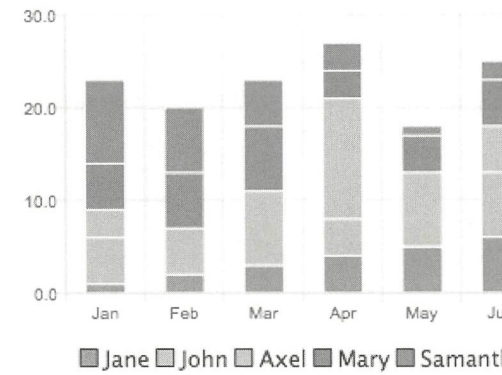
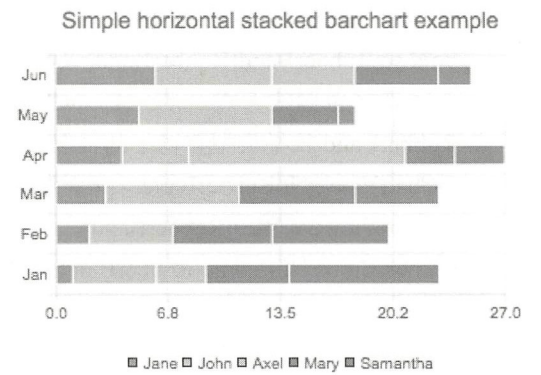
#### 2. 序列

序列是数据的表现形式, 如图 9-2 中的两条曲线就是两个 `QLineSeries` 类型的序列。



图表的类型主要就是由序列的类型决定的，常见的图表类型有折线图、柱状图、饼图、散点图等，Qt Charts 能实现的常见图表示例及用到的序列类见表 9-1。

表 9-1 Qt Charts 常见图表及用到的序列类（图片来自 Qt 帮助文件）

<p>Simple barchart example</p>  <p>■ Jane ■ John ■ Axel ■ Mary ■ Samantha</p>	<p>Simple horizontal barchart example</p>  <p>■ Jane ■ John ■ Axel ■ Mary ■ Samantha</p>
<p>柱状图 QBarSeries</p> <p>Simple percentbarchart example</p>  <p>■ Jane ■ John ■ Axel ■ Mary ■ Samantha</p>	<p>水平柱状图 QHorizontalBarSeries</p> <p>Simple horizontal percent barchart example</p>  <p>■ Jane ■ John ■ Axel ■ Mary ■ Samantha</p>
<p>百分比柱状图 QPercentBarSeries</p> <p>Simple stackedbarchart example</p>  <p>■ Jane ■ John ■ Axel ■ Mary ■ Samantha</p>	<p>水平百分比柱状图 QHorizontalPercentBarSeries</p> <p>Simple horizontal stacked barchart example</p>  <p>■ Jane ■ John ■ Axel ■ Mary ■ Samantha</p>
<p>堆叠柱状图 QStackedBarSeries</p>	<p>水平堆叠柱状图 QHorizontalStackedBarSeries</p>

续表

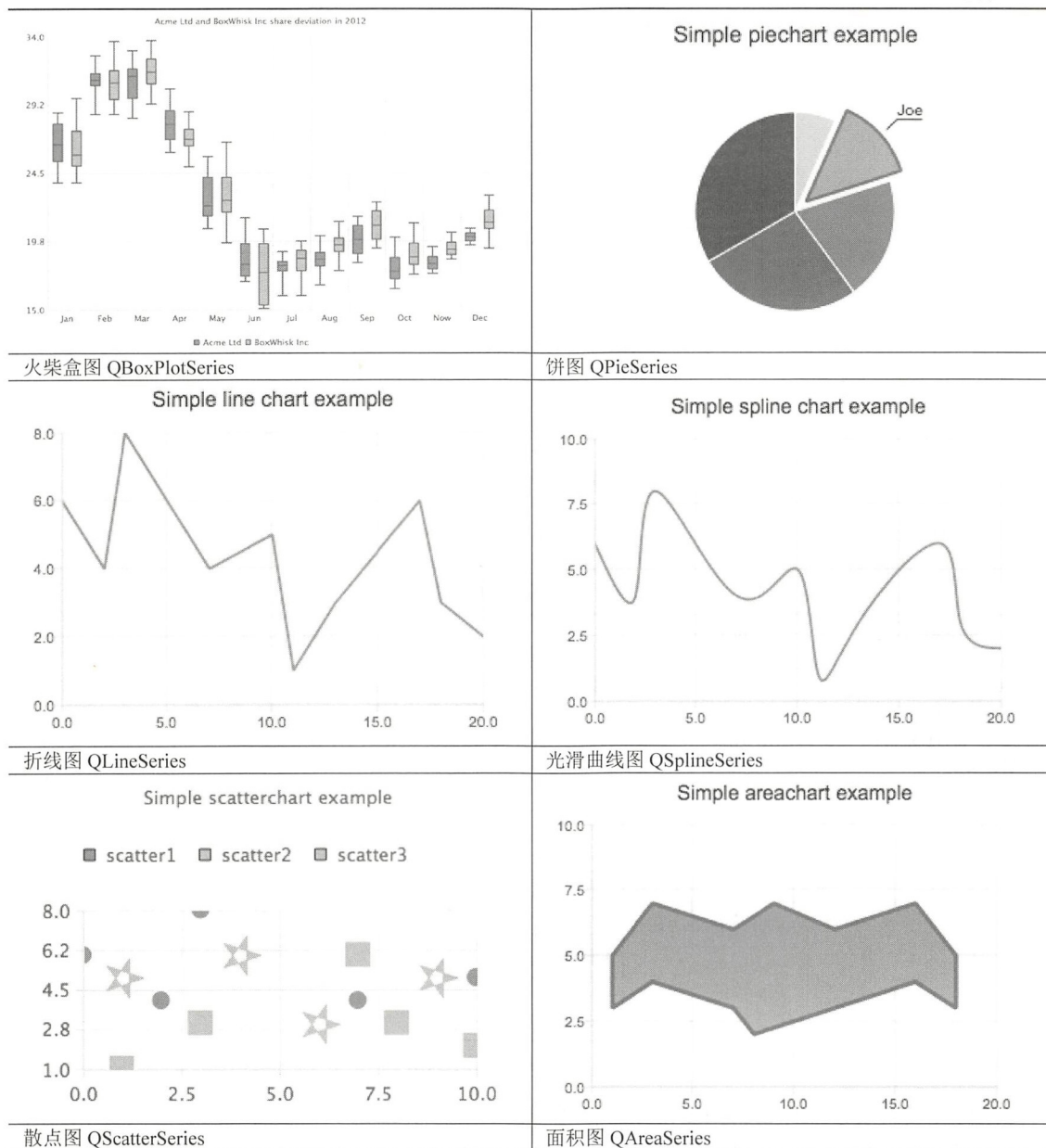


图 9-3 是这些序列类的继承关系，可见它们都是从 `QAbstractSeries` 类继承而来的。折线、光滑曲线和散点的序列是从 `QXYSeries` 继承而来，用于绘制二维平面的数据；`QAbstractBarSeries` 派生出柱状图、百分比柱状图和堆叠图等图表的序列；面积图、火柴盒图、饼图的序列都直接继承于 `QAbstractSeries`。

### 3. 坐标轴

一般的图表都有横轴和纵轴两个坐标轴，如折线图一般表示数据，坐标轴用 `QValueAxis` 类的

数值坐标轴，如果用对数坐标，就可以使用 `QLogValueAxis` 类的坐标轴；柱状图的横坐标通常是文字，可以用 `QBarCategoryAxis` 作为横轴，而饼图一般没有坐标轴。

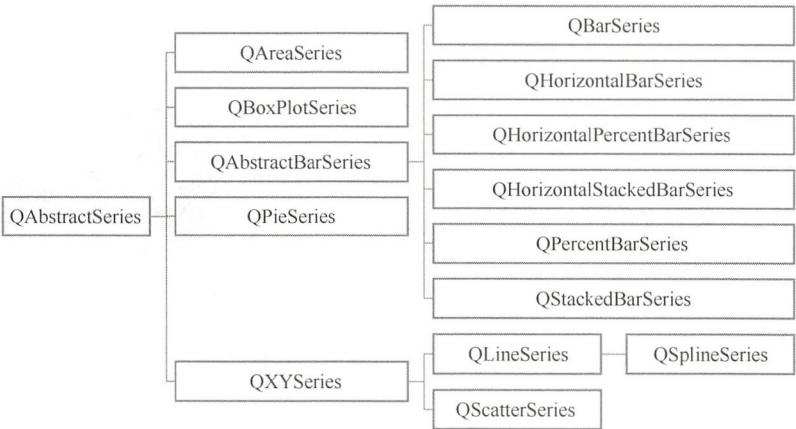


图 9-3 数据序列类的继承关系

Qt Charts 的坐标轴类、特点及其适用情况见表 9-2，类的继承关系如图 9-4 所示。

表 9-2 Qt 提供的坐标轴类

坐标轴类	特点	用途
<code>QValueAxis</code>	数值坐标轴	作为数值型数据的坐标轴
<code>QCategoryAxis</code>	分组数值坐标轴	可以为数值范围设置文字标签
<code>QLogValueAxis</code>	对数数值坐标轴	作为数值型数据的对数坐标轴，可以设置对数的基
<code>QBarCategoryAxis</code>	类别坐标轴	用字符串作为坐标轴的刻度，用于图表的非数值坐标轴
<code>QDateTimeAxis</code>	日期时间坐标轴	作为日期时间数据的坐标轴

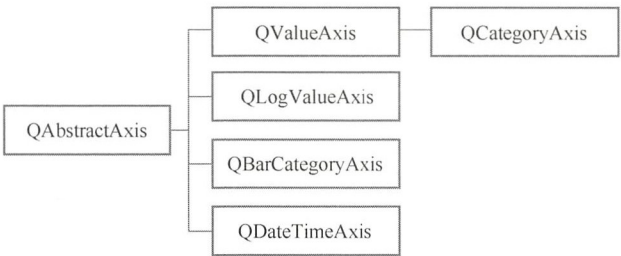


图 9-4 坐标轴类的继承关系

坐标轴类封装了坐标轴的刻度、标签、网格线、标题等属性。

#### 4. 图例

图例 (Legend) 是对图表上显示的序列的示例说明，如图 9-2 中为两条曲线显示的图例，有线条颜色和文字说明。`QLegend` 是封装了图例控制功能的类，可以为每个序列设置图例中的文字，可以控制图例显示在图表的上、下、左、右不同位置。

对于图例还有一个类 `QLegendMarker`，可以为每个序列的图例生成一个类似于 `QCheckBox` 的组件，在图例上单击序列的标记，可以控制序列是否显示。

## 9.2 QChart 绘制折线图

### 9.2.1 实例功能

实例 samp9\_2 以绘制折线图为例, 详细介绍图表各个部分的设置和操作, 包括图表的标题、图例、边距等属性设置, QLineSeries 序列的属性设置, QValueAxis 坐标轴的属性设置, 以及图表的缩放。实例运行时界面如图 9-5 所示。

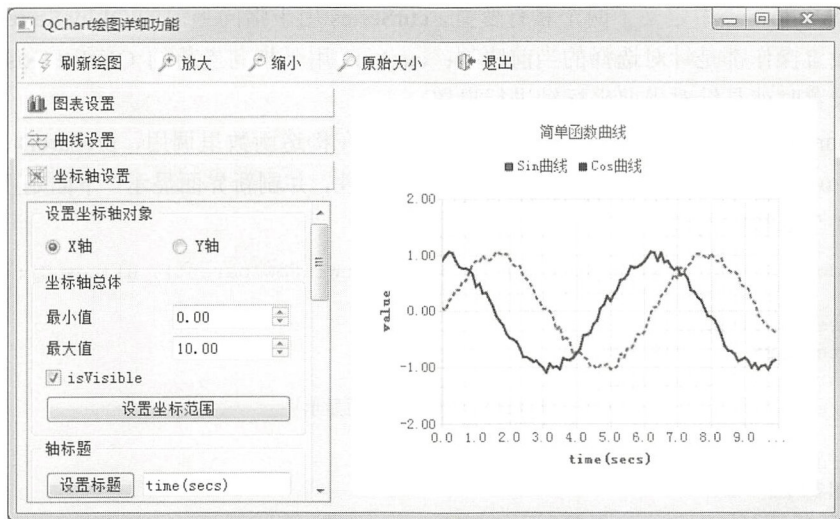


图 9-5 实例 samp9\_2 运行时界面

实例 samp9\_2 是一个主窗口继承自 QMainWindow 类的应用程序, 界面的设计主要分为以下几个部分。

- 工具栏: 创建几个 Action, 并创建工具栏, 实现图表数据刷新和缩放功能。
- 主工作区图表视图: 从组件面板放置一个 QGraphicsView 组件作为视图组件, 并用 Promote 方法升级为 QChartView 组件, 命名为 chartView。
- 图表属性设置面板: 左侧是一个 QToolBox 组件, 分为 3 个操作面板, 用于进行图表设置、曲线设置和坐标轴设置。

### 9.2.2 主窗口类定义和初始化

下面是主窗口类 MainWindow 的类定义 (省略了 Action 和界面组件的槽函数定义)。在 mainwindow.h 文件中需要包含 QtChart, 并使用宏 Qt\_CHARTS\_USE\_NAMESPACE 导入命名空间。

```
#include <QtCharts>
Qt_CHARTS_USE_NAMESPACE
class MainWindow : public QMainWindow
{
```



```

    Q_OBJECT
private:
    QLineSeries *curSeries; //当前序列
    QValueAxis *curAxis; //当前坐标轴
    void createChart(); //创建图表
    void prepareData(); //更新数据
    void updateFromChart(); //从图表更新到界面
public:
    explicit MainWindow(QWidget *parent = 0);
private:
    Ui::MainWindow *ui;
};

```

在 `MainWindow` 类中定义了两个私有变量，`curSeries` 用于指向当前的 `QLineSeries` 序列，界面上对序列的设置操作都是针对选择的当前序列；`curAxis` 用于指向当前的 `QValueAxis` 坐标轴，对坐标轴进行设置时就是针对当前坐标轴进行设置。

`createChart()` 函数用于创建图表的各个基本部件，在构造函数里调用，`prepareData()` 用于更新序列的数据，`updateFromChart()` 用于读取图表的一些属性，并刷新界面显示。下面是主窗口构造函数，以及这 3 个函数的代码。

```

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    createChart(); //创建图表
    prepareData(); //生成数据
    updateFromChart(); //从图表获得属性值，刷新界面显示
}

void MainWindow::createChart()
{ //创建图表的各个部件
    QChart *chart = new QChart();
    chart->setTitle("简单函数曲线");
    ui->chartView->setChart(chart);
    ui->chartView->setRenderHint(QPainter::Antialiasing);

    QLineSeries *series0 = new QLineSeries();
    QLineSeries *series1 = new QLineSeries();
    series0->setName("Sin 曲线");
    series1->setName("Cos 曲线");
    curSeries=series0; //当前序列

    QPen pen;
    pen.setStyle(Qt::DotLine);
    pen.setWidth(2);
    pen.setColor(Qt::red);
    series0->setPen(pen); //折线序列的线条设置
    pen.setStyle(Qt::SolidLine);
    pen.setColor(Qt::blue);
    series1->setPen(pen); //折线序列的线条设置
    chart->addSeries(series0);
    chart->addSeries(series1);

    QValueAxis *axisX = new QValueAxis;
    curAxis=axisX; //当前坐标轴

```

```

axisX->setRange(0, 10); //设置坐标轴范围
axisX->setLabelFormat("%.1f"); //标签格式
axisX->setTickCount(11); //主分隔个数
axisX->setMinorTickCount(4);
axisX->setTitleText("time(secs)"); //标题
QValueAxis *axisY = new QValueAxis;
axisY->setRange(-2, 2);
axisY->setTitleText("value");
axisY->setTickCount(5);
axisY->setLabelFormat("%.2f"); //标签格式
axisY->setMinorTickCount(4);

chart->setAxisX(axisX, series0); //设置 x 坐标轴
chart->setAxisX(axisX, series1); //设置 x 坐标轴
chart->setAxisY(axisY, series0); //设置 y 坐标轴
chart->setAxisY(axisY, series1); //设置 y 坐标轴
}

void MainWindow::prepareData()
{ //为序列生成数据
QLineSeries *series0=(QLineSeries *)ui->chartView->chart()->series().at(0);
QLineSeries *series1=(QLineSeries *)ui->chartView->chart()->series().at(1);
    series0->clear(); //清除数据
    series1->clear();
    qsrand(Qtime::currentTime().second()); //随机数初始化
    qreal t=0,y1,y2,intv=0.1;
    qreal rd;
    int cnt=100;
    for(int i=0;i<cnt;i++)
    {
        rd=(qrand() % 10)-5; //随机数,-5~+5
        y1=qSin(t)+rd/50;
        series0->append(t,y1); //序列添加数据点
        rd=(qrand() % 10)-5; //随机数,-5~+5
        y2=qCos(t)+rd/50;
        series1->append(t,y2); //序列添加数据点
        t+=intv;
    }
}

void MainWindow::updateFromChart()
{ //从图表上获取数据更新界面显示
    QChart* aChart=ui->chartView->chart(); //获取 chart
    ui->editTitle->setText(aChart->title()); //图表标题
    QMargins mg=aChart->margins(); //边距
    ui->spinMarginLeft->setValue(mg.left());
    ui->spinMarginRight->setValue(mg.right());
    ui->spinMarginTop->setValue(mg.top());
    ui->spinMarginBottom->setValue(mg.bottom());
}

```

MainWindow 类的构造函数调用 3 个私有函数进行图表和界面的初始化。

createChart()函数用于创建 QChart 对象, 创建数据序列和坐标轴, 并将这些部件组合成一个完整的图表。createChart()函数的功能与实例 samp9\_1 中的函数功能类似, 只是没有为序列添加数据点。

注意 在 `MainWindow` 类中并没有定义 `QChart` 类的成员变量，后面要使用创建的图表对象，都通过 `ui->chartView->chart()` 来获得 `QChart` 对象。

`prepareData()` 函数用于为图表中的两个序列生成数据，其中使用了随机数，以便使得每次生成的数据稍有不同。

`updateFromChart()` 函数用于将图表的标题和边距信息显示到窗口界面上。

### 9.2.3 画笔设置对话框 `QWDialogPen`

在本实例中，经常需要设置一些对象的 `pen` 属性，如折线序列的 `pen` 属性，网格线的 `pen` 属性等。`pen` 属性就是一个 `QPen` 对象，设置内容主要包括线型、线宽和颜色。为使用方便，设计一个自定义对话框 `QWDialogPen`，专门用于 `QPen` 对象的属性设置。

`QWDialogPen` 是一个可视化设计的对话框，其类定义如下：

```
class QWDialogPen : public QDialog
{ //QPen 属性设置对话框
    Q_OBJECT
private:
    QPen    m_pen; //成员变量
public:
    explicit QWDialogPen(QWidget *parent = 0);
    ~QWDialogPen();
    void    setPen(QPen pen); //设置 QPen, 用于对话框的界面显示
    QPen    getPen(); //获取对话框设置的 QPen 的属性
    static QPen    getPen(QPen iniPen, bool &ok); //静态函数
private slots:
    void on_btnColor_clicked();
private:
    Ui::QWDialogPen *ui;
};
```

自定义对话框的设计在第6章已经介绍过，这个 `QWDialogPen` 的特别之处在于定义了一个静态函数 `getPen()`，在类中的定义如下：

```
static QPen    getPen(QPen iniPen, bool &ok); //静态函数
```

Qt 的标准对话框一般都有静态函数，使用静态函数无需管理对话框的创建与删除，使用起来比较方便。

`QWDialogPen` 类的静态函数 `getPen()` 以及相关函数的实现代码如下：

```
QPen QWDialogPen::getPen(QPen iniPen, bool &ok)
{ //静态函数，获取 QPen
    QWDialogPen *Dlg=new QWDialogPen; //创建一个对话框
    Dlg->setPen(iniPen); //设置初始化 QPen
    QPen    pen;
    int ret=Dlg->exec(); //弹出对话框
    if (ret==QDialog::Accepted)
    {
        pen=Dlg->getPen(); //获取
        ok=true;    }
```

```

else
{
    pen=iniPen;
    ok=false; }
delete Dlg; //删除对话框对象
return pen; //返回设置的 QPen 对象
}

void QWDialogPen::setPen(QPen pen)
{ //设置 QPen, 并刷新显示界面
    m_pen=pen;
    ui->spinWidth->setValue(pen.width()); //线宽
    int i=static_cast<int>(pen.style()); //枚举类型转换为整型
    ui->comboPenStyle->setCurrentIndex(i);
    QColor color=pen.color();
    ui->btnColor->setAutoFillBackground(true); //设置颜色按钮的背景色
    QString str=QString::asprintf("background-color: rgb(%d, %d, %d);",
                                   color.red(),color.green(),color.blue());
    ui->btnColor->setStyleSheet(str);
}

QPen QWDialogPen::getPen()
{ //获得设置的属性
    m_pen.setStyle(Qt::PenStyle(ui->comboPenStyle->currentIndex())); //线型
    m_pen.setWidth(ui->spinWidth->value()); //线宽
    QColor color=ui->btnColor->palette().color(QPalette::Button);
    m_pen.setColor(color); //颜色
    return m_pen;
}

```

静态函数 `getPen()` 里创建了一个 `QWDialogPen` 类的实例 `Dlg`, 然后调用 `Dlg->setPen(iniPen)` 进行初始化设置, 运行对话框并获取返回状态, 若返回类型为 `QDialog::Accepted`, 就调用 `Dlg->getPen()` 获取设置属性后的 `QPen` 的对象, 最后删除对话框对象并返回设置的 `QPen` 对象。所以, 静态函数 `getPen()` 就是集成了普通方法调用对话框时的创建对话框、设置初始值、获取对话框返回状态、获取返回值、删除对话框的过程, 简化了调用代码。

在设置颜色按钮的背景色时用到了 `StyleSheet` 的功能, 这一点在 16.2 节有详细介绍。

图 9-6 是 `QWDialogPen` 对话框运行界面, 可以选择线型、设置线宽和颜色。

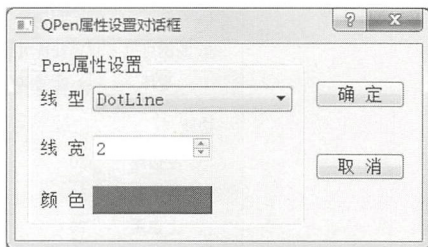


图 9-6 QPen 设置对话框运行状态

## 9.2.4 QChart 的设置

`QChart` 是组合图表各部件、显示各种数据序列的绘图组件。`QChart` 接口函数众多, 其主要接口函数分类整理后见表 9-3。对于一个属性, 通常有一个设置函数和一个对应的读取函数, 如 `setTitle()` 用于设置图表标题, 对应的读取图表标题的函数为 `title()`。表 9-3 仅列出设置函数或单独地读取函数, 仅列出函数的返回数据类型, 省略了输入参数, 函数详细定义请参考 Qt 帮助文件。



表 9-3 QChart 类的主要函数

分组	函数名	功能描述
图表外观	void setTitle()	设置图表标题，显示在图表上方，支持 HTML 格式
	void setTitleFont()	设置图表标题字体
	void setTitleBrush()	设置图表标题画刷
	void setTheme()	设置主题，主题是内置的 UI 设置，定义了图表的配色
	void setMargins()	设置绘图区与图表边界的 4 个边距
	QLegend * legend()	返回图表的图例，是一个 QLegend 类的对象
	void setAnimationOptions()	设置序列或坐标轴的动画效果
数据序列	void addSeries()	添加序列
	QList<QAbstractSeries *> series()	返回图表拥有的序列的列表
	void removeSeries()	移除一个序列，但并不删除序列对象
	void removeAllSeries()	移除并删除图表的所有序列
坐标轴	void addAxis()	为图表的某个方向添加坐标轴
	QList<QAbstractAxis *> axes()	返回某个方向的坐标轴列表
	void setAxisX()	设置某个序列的水平方向的坐标轴
	void setAxisY()	设置某个序列的垂直方向的坐标轴
	void removeAxis()	移除一个坐标轴
	void createDefaultAxes()	根据已添加的序列的类型，创建缺省的坐标轴，前面已有的坐标轴会被删除

图 9-7 是进行图表设置的界面，通过界面可以设置图表标题的文字内容和字体，可以设置图例的位置、是否显示、字体和颜色，可以设置 4 个边距的值，可以设置动画效果以及主题。图 9-8 是进行曲线序列设置的界面，可操作内容如下。

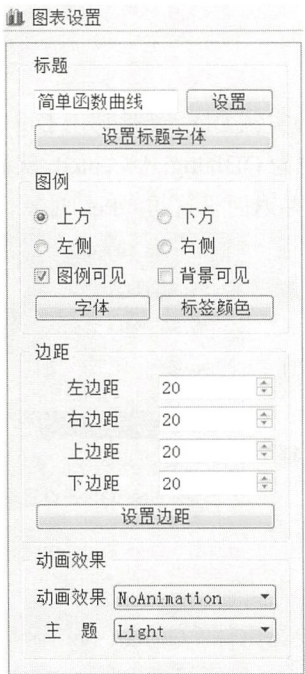


图 9-7 图表设置界面



图 9-8 曲线序列设置界面

setAnimationOptions(AnimationOptions options)函数设置图表的动画效果，输入参数是 QChart::AnimationOption 枚举类型，有以下几种取值：

- QChart::NoAnimation——无动画效果；
- QChart::GridAxisAnimations——背景网格有动画效果；
- QChart::SeriesAnimations——序列有动画效果；
- QChart::AllAnimations——都有动画效果。

主题是预定义的图表配色样式，是 QChart::ChartTheme 枚举类型，有多种取值，使图表具有不同的配色效果。

图例是一个 QLegend 类的对象，通过 QChart::legend() 可以获得图表的图例。图例是根据添加的序列自动生成的，但是可以修改图例的一些属性，如在图表中的显示位置、图例文字的字体等。例如，设置图例显示在图表的底部可用下面的语句：

```
ui->chartView->chart()->legend()->setAlignment(Qt::AlignBottom);
```

设置图例文字的字体的代码如下：

```
void MainWindow::on_btnLegendFont_clicked()
{ //图例的字体设置
    QFont font=ui->chartView->chart()->legend()->font();
    bool ok=false;
    font=QFontDialog::getFont(&ok,font);
    if (ok)
        ui->chartView->chart()->legend()->setFont(font);
}
```

### 9.2.5 QLineSeries 序列的设置

本实例图表的序列使用的是 QLineSeries，它是 QXYSeries 的子类，用于绘制二维数据点的折线图。QLineSeries 的主要函数见表 9-4（包括从父类继承的函数，仅列出函数的返回数据类型，省略了输入参数）。

表 9-4 QLineSeries 类的主要函数

分组	函数	功能描述
序列名称	void setName()	设置序列的名称，这个名称会显示在图例里，支持 HTML 格式
图表	QChart* chart()	返回序列所属的图表对象
序列外观	void setVisible()	设置序列可见性
	void show()	显示序列，使序列可见
	void hide()	隐藏序列，使其不可见
	void setColor()	设置序列线条的颜色
	void setPen()	设置绘制线条的画笔
	void setBrush()	设置绘制数据点的画刷
	void setOpacity()	设置序列的透明度，0 表示完全透明，1 表示不透明
数据点	void setPointsVisible()	设置数据点的可见性
	void append()	添加一个数据点到序列
	void insert()	在某个位置插入一个数据点
	void replace()	替换某个数据点
	void clear()	清除所有数据点
	void remove()	删除某个数据点
	void removePoints()	从某个位置开始，删除指定个数的数据点
	int count()	数据点的个数

续表

分组	函数	功能描述
数据点	QPointF& at()	返回某个位置的数据点
	QList<QPointF> points()	返回数据点的列表
	QVector<QPointF> pointsVector()	返回数据点的, 效率更高
数据点标签	void setPointLabelsVisible()	设置数据点标签的可见性
	void setPointLabelsColor()	设置数据点标签的文字颜色
	void setPointLabelsFont()	设置数据点标签字体
	void setPointLabelsFormat()	设置数据点标签格式
	void setPointLabelsClipping()	设置标签的裁剪属性, 缺省为 True, 即绘图区外的标签被裁剪掉
坐标轴	bool attachAxis()	为序列附加一个坐标轴, 通常需要一个 X 轴和一个 Y 轴
	bool detachAxis()	解除一个附加的坐标轴
	QList<QAbstractAxis *> attachedAxes()	返回附加的坐标轴的列表

实例中对曲线序列进行属性设置的界面如图 9-8 所示。首先通过上方的“选择操作序列”里的两个 RadioButton 按钮选择当前操作序列, 两个按钮的代码相同, 代码如下:

```
void MainWindow::on_radioSeries0_clicked()
{
    // 获取当前数据序列
    if (ui->radioSeries0->isChecked())
        curSeries=(QLineSeries *)ui->chartView->chart()->series().at(0);
    else
        curSeries=(QLineSeries *)ui->chartView->chart()->series().at(1);
    // 获取序列的属性值, 并显示到界面上
    ui->editSeriesName->setText(curSeries->name());
    ui->chkSeriesVisible->setChecked(curSeries->isVisible());
    ui->chkPointVisible->setChecked(curSeries->pointsVisible());
    ui->sliderSeriesOpacity->setValue(curSeries->opacity()*10);
    ui->chkPointLabelVisible->setChecked(curSeries->pointLabelsVisible());
}
```

curSeries 是在 MainWindow 类里定义的私有变量, 用于指向当前操作的序列。选择序列后, 会将当前序列的一些属性值显示到界面上, 如序列名称、序列可见性、序列数据点可见性等。

序列的一些常规属性的设置都很简单, 例如序列名称设置、序列可见性设置、曲线 pen 属性设置等, 调用相应函数即可。

数据点标签的格式设置使用函数 setPointLabelsFormat(), 有两种数据可以在数据点标签中显示, 有固定的标签:

@xPoint, 数据点的 X 值;

@yPoint, 数据点的 Y 值。

例如, 使数据点标签只显示 Y 值, 设置语句为:

```
curSeries->setPointLabelsFormat("@yPoint");
```

如果使数据点标签显示(X, Y)值, 设置语句为:

```
curSeries->setPointLabelsFormat("@xPoint,@yPoint");
```

为一个序列添加数据点, 可以使用 append()函数, 也可以使用流操作符“<<”, 如 prepareData()函数中添加数据点的部分也可以改写为:

```
*series0<<QPointF(t,y1); //序列添加数据点
```

为序列指定坐标轴, 在前面的 createChart()函数中, 使用 QChart 的函数为序列设置坐标轴:



```
chart->setAxisX(axisX, series0); //添加 X 坐标轴
chart->setAxisX(axisX, series1); //添加 X 坐标轴
chart->setAxisY(axisY, series0); //添加 Y 坐标轴
chart->setAxisY(axisY, series1); //添加 Y 坐标轴
```

QChart::setAxisX()函数为序列指定 X 坐标轴，并将坐标轴添加到图表里；QChart::setAxisY()函数为序列指定 Y 坐标轴，并将坐标轴添加到图表里。无需再调用序列的 attachAxis()函数。若要使用序列的 attachAxis()函数，则实现上述功能的代码如下：

```
chart->addAxis(axisX, Qt::AlignBottom); //坐标轴添加到图表，并指定方向
chart->addAxis(axisY, Qt::AlignLeft);
series0->attachAxis(axisX); //序列 series0 附加坐标轴
series0->attachAxis(axisY);
series1->attachAxis(axisX); //序列 series1 附加坐标轴
series1->attachAxis(axisY);
```

即先用 QChart::addAxis()函数添加一个坐标轴到图表，并指定坐标轴的方向，然后用序列的 attachAxis()函数附加坐标轴对象。

### 9.2.6 QValueAxis 坐标轴的设置

本例中使用 QValueAxis 类的坐标轴，这是数值型坐标轴，与 QLineSeries 正好配合使用。Qt 提供了好几种坐标轴类（见图 9-4），都是从 QAbstractAxis 类继承而来的。QValueAxis 类的主要函数见表 9-5（包括从 QAbstractAxis 继承的函数，仅列出函数的返回数据类型，省略了输入参数）。

表 9-5 QValueAxis 类的主要函数

分组	函数	功能描述
坐标轴整体	void setVisible()	设置坐标轴可见性
	Qt::Orientation orientation()	返回坐标轴方向
	void setMin()	设置坐标轴最小值
	void setMax()	设置坐标轴最大值
	void setRange()	设置坐标轴最小、最大值表示的范围
轴标题	void setTitleVisible()	设置轴标题的可见性
	void setTitleText()	设置轴标题的文字
	void setTitleFont()	设置轴标题的字体
	void setTitleBrush()	设置轴标题的画刷
轴标签	void setLabelFormat()	设置标签格式，例如可以设置显示的小数点位数
	void setLabelsAngle()	设置标签的角度，单位为度
	void setLabelsBrush()	设置标签的画刷
	void setLabelsColor()	设置标签文字颜色
	void setLabelsFont()	设置标签文字字体
	void setLabelsVisible()	设置轴标签文字是否可见
轴线和刻度线	void setTickCount()	设置坐标轴主刻度的个数
	void setLineVisible()	设置轴线和刻度线的可见性
	void setLinePen()	设置轴线和刻度线的画笔
	void setLinePenColor()	设置轴线和刻度线的颜色
主网格线	void setGridLineColor()	设置网格线的颜色
	void setGridLinePen()	设置网格线的画笔
	void setGridLineVisible()	设置网格线的可见性



续表

分组	函数	功能描述
次刻度和次网格线	void setMinorTickCount()	设置两个主刻度之间的次刻度的个数
	void setMinorGridLineColor()	设置次网格线的颜色
	void setMinorGridLinePen()	设置次网格线的画笔
	void setMinorGridLineVisible()	设置次网格线的可见性

参看图 9-5 图表的  $X$  坐标轴， $QValueAxis$  坐标轴有以下几个组成部分。

- 坐标轴标题：是在坐标轴下方显示的文字，表示坐标轴的名称，图中  $X$  轴坐标轴的标题是“time(secs)”。坐标轴标题除了可以设置文字内容，还可以设置字体、画刷和可见性。
- 轴线和刻度线：轴线是图中从左到右的表示坐标轴的直线，刻度线是垂直于轴线的短线，包括主刻度线和次刻度线，主刻度个数是 `tickCount()`，每两个主刻度之间的次刻度的个数是 `minorTickCount()`。
- 轴标签：在主刻度处显示的数值标签文字，可以控制其数值格式、文字颜色和字体等。
- 主网格线：在绘图区与主刻度对应的网格线，可以设置其颜色、线条的 `pen` 属性、可见性等。
- 次网格线：在绘图区与次刻度对应的网格线，可以设置其颜色、线条的 `pen` 属性、可见性等。

搞清楚坐标轴的这些组成部分后，对其进行属性读取或设置就只需调用相应的函数即可。图 9-9 是实例中窗口左侧“坐标轴设置”的界面内容，可以对坐标轴的各种属性进行设置。



图 9-9 坐标轴设置的界面

在图 9-9 的界面上首先选择需要操作的坐标轴对象，两个 `RadioButton` 按钮的响应代码相同，为 `radioX` 按钮编写事件槽函数，另一个按钮的槽函数里只需调用这个函数即可，代码如下：

```
void MainWindow::on_radioX_clicked()
{ // 获取当前坐标轴
    if (ui->radioX->isChecked())
```

```

        curAxis=(QValueAxis*)ui->chartView->chart()->axisX();
    else
        curAxis=(QValueAxis*)ui->chartView->chart()->axisY();
//获取坐标轴的各种属性，显示到界面上
    ui->spinAxisMin->setValue(curAxis->min());
    ui->spinAxisMax->setValue(curAxis->max());
    ui->editAxisTitle->setText(curAxis->titleText());
    ui->chkBoxAxisTitle->setChecked(curAxis->isTitleVisible());
    ui->editAxisLabelFormat->setText(curAxis->labelFormat());
    ui->chkBoxLabelsVisible->setChecked(curAxis->labelsVisible());
    ui->chkGridLineVisible->setChecked(curAxis->isGridLineVisible());
    ui->chkAxisLineVisible->setChecked(curAxis->isLineVisible());
    ui->spinTickCount->setValue(curAxis->tickCount());
    ui->chkAxisLineVisible->setChecked(curAxis->isLineVisible());
    ui->spinMinorTickCount->setValue(curAxis->minorTickCount());
    ui->chkMinorTickVisible->setChecked(curAxis->isMinorGridLineVisible());
}

```

程序中 `curAxis` 是在 `MainWindow` 类中定义的私有变量，用于表示当前操作的坐标轴对象。获取对象后，首先获得轴对象的各种属性并显示在界面上。

坐标轴各种属性的设置只需调用 `QValueAxis` 的相应函数即可，各种接口函数可参考表 9-5。例如，“标签格式”按钮的槽函数代码如下：

```

void MainWindow::on_pushButton_clicked()
{
    //设置坐标轴刻度标签的文字格式
    curAxis->setLabelFormat(ui->editAxisLabelFormat->text());
}

```

在编辑框 `editAxisLabelFormat` 里设置格式字符串，例如 “%.2f”，作为 `QValueAxis::setLabelFormat()` 的输入参数。格式字符串的定义与 `printf()` 函数的格式字符串定义一样。

## 9.3 各种常见图表的绘制

### 9.3.1 实例功能概述

前一节通过折线图的绘制，介绍了 Qt Charts 绘图的基本原理。图表的类型由数据序列决定，除了折线图，Qt Charts 还提供柱状图、饼图、百分比柱状图等常见图表。

本节通过实例 `samp9_3` 介绍这些常见图表的绘制方法，图 9-10 是实例 `samp9_3` 的运行界面。左上方是随机生成的若干个学生的数学、语文、英语分数，平均分是自动计算的，左下方是根据分数统计的结果，右方是各种图表的页面。

该实例的主要目的是演示各种常见图表的绘制方法，图表的基本设置在程序里完成，不再如实例 `samp9_2` 那样提供丰富的设置功能。

窗口左上方是一个 `QTableView` 组件，使用 `Model/View` 结构提供数据编辑功能；左下角是一个 `QTreeWidget` 组件，用于显示统计数据结果；窗口右方是一个 `QTabWidget` 组件，共有 5 个页面，分别用于显示柱状图、饼图、堆叠图、百分比柱状图和散点图，每个图表的显示用一个 `Qgraphics View` 组件升级为 `QChartView`。



图 9-10 实例 samp9\_3 运行界面

### 9.3.2 数据准备

图 9-10 左上方的数据采用 Model/View 结构，在主窗口类 MainWindow 里定义了数据模型，并在主窗口的构造函数里初始化数据。下面是 MainWindow 的类定义（省略了部分窗口界面组件的槽函数定义）。

```
#define    fixedColumnCount    5    //数据模型的列数
#define    iniDataRowCount    6    //学生个数
#define    colNoName          0    //姓名的列编号
#define    colNoMath          1    //数学的列编号
#define    colNoChinese       2    //语文的列编号
#define    colNoEnglish       3    //英语的列编号
#define    colNoAverage       4    //平均分的列编号
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QStandardItemModel *theModel; //数据模型
    void    iniData();             //初始化数据
    void    surveyData();          //统计数据

    void    iniBarChart();         //柱状图初始化
    void    buildBarChart();       //构建柱状图
    void    iniPieChart();         //饼图
    void    buildPieChart();       //构建饼图
    void    iniStackedBar();       //堆叠图
    void    buildStackedBar();     //构建堆叠图
    void    iniPercentBar();       //百分比柱状图
    void    buildPercentBar();     //构建百分比柱状图
    void    iniScatterChart();     //散点图
    void    buildScatterChart();   //构建散点图
```



```
private slots:
//三个分数列的数据发生变化,重新计算平均分
void on_itemChanged(QStandardItem *item);
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    Ui::MainWindow *ui;
};
```

在头文件中首先定义了一些常量,包括数据模型的列数 `fixedColumnCount` 和学生人数 `iniDataRowCount`。定义为常量后便于修改,例如现在学生人数定义为 6 个,后续为了增大数据样本,可以将学生人数定义为 30 人。头文件里还定义了数据模型中 5 个列的编号,在后续的程序里使用常量。

`MainWindow` 类中定义了数据模型类变量 `theModel`,函数 `iniData()`用于随机初始化数据,`surveyData()`用于对数据进行统计。

每个图表有两个函数,一个用于初始化,一个用于根据数据构建图表。如 `iniBarChart()`用于初始化柱状图,`buildBarChart()`用于根据数据构建柱状图。

数据模型中每名学生的平均分根据 3 门课的成绩自动计算,希望在界面修改一个分数时,自动计算其平均分。这里可以利用 `QStandardItemModel` 的 `itemChanged()`信号,在某个数据项被修改后会发射此信号,定义了一个私有槽函数 `on_itemChanged()`用于响应此信号,实现平均分的自动计算。

下面是 `MainWindow` 的构造函数的实现代码。

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    theModel = new QStandardItemModel(iniDataRowCount, fixedColumnCount, this);
    iniData(); //初始化数据
    surveyData(); //数据统计
//数据模块的 itemChanged 信号与自定义的槽函数关联,用于自动计算平均分
    connect(theModel, SIGNAL(itemChanged(QStandardItem *)),
            this, SLOT(on_itemChanged(QStandardItem *)));
    ui->tableView->setModel(theModel); //设置数据模型
    iniBarChart(); //柱状图初始化
    iniPieChart(); //饼图初始化
    iniStackedBar(); //堆叠图初始化
    iniPercentBar(); //百分比柱状图初始化
    iniScatterChart(); //散点图初始化
}
```

在构造函数中,创建数据模型 `theModel`,然后初始化数据、统计数据,将数据模型的 `itemChanged()`信号与自定义槽函数 `on_itemChanged()`关联起来。然后对 5 个图表调用函数进行初始化。

下面是 `iniData()`函数的实现代码,用于对数据模型进行初始化。

```
void MainWindow::iniData()
{ //数据初始化
    QStringList headerList;
    headerList<<"姓名"<<"数学"<<"语文"<<"英语"<<"平均分";
    theModel->setHorizontalHeaderLabels(headerList); //设置表头文字
    qsrand(Qtime::currentTime().second()); //随机数种子
```



```

for (int i=0;i<theModel->rowCount();i++)
{
    QString studName=QString::asprintf("学生%2d",i+1);
    QStandardItem* aItem=new QStandardItem(studName);//创建 item
    aItem->setTextAlignment(Qt::AlignHCenter);
    theModel->setItem(i,colNoName,aItem); //学生列, 设置 item
    qreal avgScore=0;
    for (int j=colNoMath;j<=colNoEnglish;j++) //数学, 语文, 英语
    { //不包含最后一列
        qreal score=50.0+(qrand() % 50);//随机数
        avgScore+=score;
        aItem=new QStandardItem(QString::asprintf("%.0f",score));
        aItem->setTextAlignment(Qt::AlignHCenter);
        theModel->setItem(i,j,aItem); //设置 Item
    }
    aItem=new QStandardItem(QString::asprintf("%.1f",avgScore/3));//平均分
    aItem->setTextAlignment(Qt::AlignHCenter);
    aItem->setFlags(aItem->flags() & (!Qt::ItemIsEditable)); //不允许编辑
    theModel->setItem(i,colNoAverage,aItem);
}
}

```

数据初始化会设置表头标题, 数学、语文、英语分数采用随机数生成, 然后计算平均分。其中, 平均分一列是不允许编辑的。

数据统计函数 `surveyData()` 用于对数据模型里的数据按照分数段进行统计, 然后显示在 `QTreeWidget` 组件里。数据统计是简单的数学计算, 对 `QTreeWidget` 的操作在 4.7 节有介绍, 所以 `surveyData()` 函数的代码不再具体显示。

自定义槽函数 `on_itemChanged()` 用于在分数修改后自动计算平均分, 下面是槽函数 `on_itemChanged()` 的代码。

```

void MainWindow::on_itemChanged(QStandardItem *item)
{ //自动计算平均分
    if ((item->column()<colNoMath) || (item->column()>colNoEnglish))
        return; //如果被修改的 item 不是数学、语文、英语, 就退出
    int rowNo=item->row(); //获取数据的行编号
    qreal avg=0;
    QStandardItem *aItem;
    for (int i=colNoMath;i<=colNoEnglish;i++)
    { //获取三个分数的和
        aItem=theModel->item(rowNo,i);
        avg+=aItem->text().toDouble();
    }
    avg=avg/3; //计算平均分
    aItem=theModel->item(rowNo,colNoAverage); //获取平均分数据的 item
    aItem->setText(QString::asprintf("%.1f",avg));
}

```

传递的参数 `item` 是被修改的数据项。程序首先判断该数据项是不是 3 个基本分数, 如果不是就退出。通过 `item->row()` 获取当前数据的行号, 然后提取 3 个基本分数计算平均分, 再更新数据模型中的平均分数据项。

### 9.3.3 柱状图

#### 1. 柱状图的绘制

MainWindow 的构造函数里调用 `iniBarChart()` 函数对绘制柱状图进行初始化, 单击图 9-10 的“BarChart”页面的“刷新柱状图”按钮, 会调用 `buildBarChart()` 绘制柱状图。绘制的柱状图如图 9-10 所示, 柱状图由 3 个数据集组成, 分别是数学、语文和英语, 平均分用折线序列显示。图表的横坐标是学生姓名, 纵坐标是数值。

`iniBarChart()` 函数的代码如下:

```
void MainWindow::iniBarChart()
{ //柱状图初始化
    QChart *chart = new QChart();
    chart->setTitle("BarChart 演示");
    chart->setAnimationOptions(QChart::SeriesAnimations);
    ui->chartViewBar->setChart(chart);
    ui->chartViewBar->setRenderHint(QPainter::Antialiasing);
}
```

在可视化设计界面时, 放置了一个 `QGraphicsView` 组件, 然后升级为 `QChartView`, 并命名为 `chartViewBar`。在 `iniBarChart()` 函数的代码里, 创建了一个 `QChart` 对象, 然后设置为 `QChartView` 组件显示的图表。

`buildBarChart()` 函数用于绘制柱状图和平均分折线图, 代码如下:

```
void MainWindow::buildBarChart()
{ //构造柱状图
    QChart *chart = ui->chartViewBar->chart(); //获取 ChartView 关联的 chart
    chart->removeAllSeries(); //删除所有序列
    chart->removeAxis(chart->axisX()); //删除坐标轴
    chart->removeAxis(chart->axisY()); //删除坐标轴
    //创建三个 QBarSet 数据集, 从数据模型的表头获取 Name
    QBarSet *setMath = new QBarSet(theModel->horizontalHeaderItem(colNoMath)->text());
    QBarSet *setChinese = new QBarSet(theModel->horizontalHeaderItem(colNoChinese)->text());
    QBarSet *setEnglish = new QBarSet(theModel->horizontalHeaderItem(colNoEnglish)->text());
    QLineSeries *Line = new QLineSeries(); //用于显示平均分
    Line->setName(theModel->horizontalHeaderItem(colNoAverage)->text());
    QPen pen;
    pen.setColor(Qt::red);
    pen.setWidth(2);
    Line->setPen(pen);

    for(int i=0; i<theModel->rowCount(); i++)
    { //从数据模型获取数据
        setMath->append(theModel->item(i, colNoMath)->text().toInt());
        setChinese->append(theModel->item(i, colNoChinese)->text().toInt());
        setEnglish->append(theModel->item(i, colNoEnglish)->text().toInt());
        Line->append(QPointF(i, theModel->item(i, colNoAverage)->text().toFloat()));
    }
    //创建一个柱状图序列 QBarSeries, 并添加三个数据集
    QBarSeries *series = new QBarSeries();
    series->append(setMath);
```

```

series->append(setChinese);
series->append(setEnglish);
chart->addSeries(series); //添加柱状图序列
chart->addSeries(Line); //添加折线图序列
//用于横坐标的字符串列表,即学生姓名
QStringList categories;
for (int i=0;i<theModel->rowCount();i++)
    categories <<theModel->item(i,colNoName)->text();
//用于柱状图的横坐标轴
QBarCategoryAxis *axisX = new QBarCategoryAxis();
axisX->append(categories); //添加横坐标文字列表
chart->setAxisX(axisX, series); //设置横坐标
chart->setAxisX(axisX, Line); //设置横坐标
axisX->setRange(categories.at(0), categories.at(categories.count()-1));
//数值型坐标作为纵坐标轴
QValueAxis *axisY = new QValueAxis;
axisY->setRange(0, 100);
axisY->setTitleText("分数");
axisY->setTickCount(6);
axisY->setLabelFormat("%.0f"); //标签格式
chart->setAxisY(axisY, series);
chart->setAxisY(axisY, Line);
chart->legend()->setVisible(true); //显示图例
chart->legend()->setAlignment(Qt::AlignBottom); //图例显示在下方
}

```

代码里首先通过下面的一行语句获取 `chartViewBar` 关联的 `QChart` 组件, 然后对 `Chart` 的操作就都可以使用此变量。

```
QChart *chart =ui->chartViewBar->chart();
```

获得 `chart` 之后, 先删除 `chart` 原有的所有序列和坐标轴。

一个柱状图由多个数据集组成, 在此创建了 3 个 `QBarSet` 数据集, 分别对应数学、语文、英语 3 门课的分。

柱状图的数据集类是 `QBarSet`, 创建对象时使用了数据模型的列标题作为数据集的名称, 这个名称会显示在图例里。

显示平均分使用 `QLineSeries` 序列, 即折线序列。

创建 3 个 `QBarSet` 数据集和折线序列后, 遍历数据模型将数据添加到数据集或折线序列。

`QBarSet::append(const qreal value)` 只需添加一个数值, 如:

```
setMath->append(theModel->item(i,colNoMath)->text().toInt()); //数学
```

`QLineSeries::append(const QPointF &point)` 则是添加一个数据点。

为 3 个 `QBarSet` 数据集添加完数据后, 创建一个 `QBarSeries` 序列, 并将 3 个数据集添加到这个 `QBarSeries` 序列, 即:

```

QBarSeries *series = new QBarSeries();
series->append(setMath);
series->append(setChinese);
series->append(setEnglish);

```

所以, 虽然是有 3 个数据集, 但是只有一个柱状图序列。然后将 `QBarSeries` 序列和 `QLineSeries`

序列添加到图表。

添加完序列后要创建坐标轴。横坐标是 `QBarCategoryAxis` 类的坐标，是每个学生的姓名；纵坐标是 `QValueAxis` 类型坐标，是数值型坐标，为其设置范围和刻度等属性后，设置为两个序列的 *Y* 轴坐标。

## 2. 柱状图相关的主要类

从以上创建柱状图的程序可以看出，与柱状图相关的主要类包括以下 3 个。

- `QBarSet`：用于创建柱状图的数据集。
- `QBarSeries`：柱状图序列，一个柱状图序列一般包含多个 `QBarSet` 数据集。
- `QBarCategoryAxis`：柱状图分类坐标，以文字标签形式表示的坐标。

`QBarSet` 是直接 from `QObject` 继承而来的类，其主要的函数功能见表 9-6（仅列出函数的返回数据类型，省略了输入参数）。

表 9-6 `QBarSet` 类的主要函数功能

分组	函数	功能描述
标签	<code>void setLabel()</code>	设置数据集的标签，用于图例显示的文字
	<code>void setLabelBrush()</code>	设置标签的画刷
	<code>void setLabelColor()</code>	设置标签的文字颜色
	<code>void setLabelFont()</code>	设置标签的字体
数据棒	<code>void setBorderColor()</code>	设置数据集的棒图的边框颜色
	<code>void setBrush()</code>	设置数据集的棒图的画刷
	<code>void setColor()</code>	设置数据集的棒图填充颜色
	<code>void setPen()</code>	设置数据集的棒图的边框画笔
数据点	<code>void append()</code>	添加一个数据到数据集
	<code>void insert()</code>	在某个位置插入一个数据到数据集
	<code>void remove()</code>	从某个位置开始删除一定数量的数据
	<code>void replace()</code>	替换某个位置的数据
	<code>qreal at()</code>	返回某个位置的数据
	<code>int count()</code>	返回数据的个数
	<code>qreal sum()</code>	返回数据集内所有数据的和

`QBarSeries` 是绘制柱状图序列的类，从 `QAbstractBarSeries` 类继承而来。`QBarSeries` 类主要实现对 `QBarSet` 数据集的操作，一个柱状图序列可以有多个数据集。`QBarSeries` 类的主要函数功能见表 9-7（包括从 `QAbstractBarSeries` 继承的函数，仅列出函数的返回数据类型，省略了输入参数）。

表 9-7 `QBarSeries` 类的主要函数功能

分组	函数	功能描述
外观	<code>void setBarWidth()</code>	设置数据棒的宽度
	<code>void setLabelsVisible()</code>	设置数据棒的标签可见性
	<code>void setLabelsFormat()</code>	设置数据棒的标签的格式，只支持一种： <code>@value</code>
	<code>void setLabelsPosition()</code>	数据棒标签的位置，可在数据棒的中间、顶端、底端、外部
	<code>void setLabelsAngle()</code>	设置标签的角度
数据集	<code>bool append()</code>	添加一个 <code>QBarSet</code> 数据集到序列
	<code>bool insert()</code>	在某个位置插入一个 <code>QBarSet</code> 数据集到序列
	<code>bool remove()</code>	移除一个数据集，解除所属关系，并删除数据集对象
	<code>bool take()</code>	移除出一个数据集，但是不删除数据集对象
	<code>void clear()</code>	清除全部数据集，并删除数据集对象
	<code>QList&lt;QBarSet *&gt; barSets()</code>	返回数据集对象的列表
	<code>int count()</code>	返回数据集的个数



柱状图的横坐标一般是文字表示的类别，所以用于柱状图的横坐标类是 `QBarCategoryAxis`，表 9-8 是 `QBarCategoryAxis` 的主要函数的功能（仅列出函数的返回数据类型，省略了输入参数）。

表 9-8 `QBarCategoryAxis` 主要函数功能

分组	函数	功能描述
坐标内容	<code>void append()</code>	添加一个类别（category）到坐标轴
	<code>void insert()</code>	在某个位置插入一个类别到坐标轴
	<code>void replace()</code>	替换某个类别
	<code>void remove()</code>	移除某个类别
	<code>void clear()</code>	删除所有类别
	<code>QString at()</code>	返回某个索引位置的类别文字
	<code>int count()</code>	返回类别的个数
	<code>void setCategories()</code>	设置一个 <code>QStringList</code> 字符串列表作为坐标轴的类别文字，删除原来所有类别文字
坐标范围	<code>void setMin()</code>	设置坐标轴最小值
	<code>void setMax()</code>	设置坐标轴最大值
	<code>void setRange()</code>	设置坐标轴范围

`QBarCategoryAxis` 坐标轴的内容是一系列文字表示的类别，所以，其函数基本都是针对字符串的操作。

## 9.3.4 饼图

### 1. 饼图的绘制

图 9-11 是绘制饼图的页面。饼图一般用于对一个数据进行分段统计后分析显示，可以直观地表示某个数据所占的百分比。在图 9-11 中，绘制饼图需在上方的“分析数据”下拉列表框中选择分析数据对象，列表框中有数学、英语、语文、平均分 4 个数据对象可供选择，在下拉列表框中选择一个项，或单击“刷新饼图”按钮就可以刷新饼图。

饼图数据来源于分数的统计数据，即每个分数段的人数。为使每个分数段都有一定的统计人数，将 `mainwindow.h` 中的常量 `iniDataRowCount` 更改为较大值，如 40。

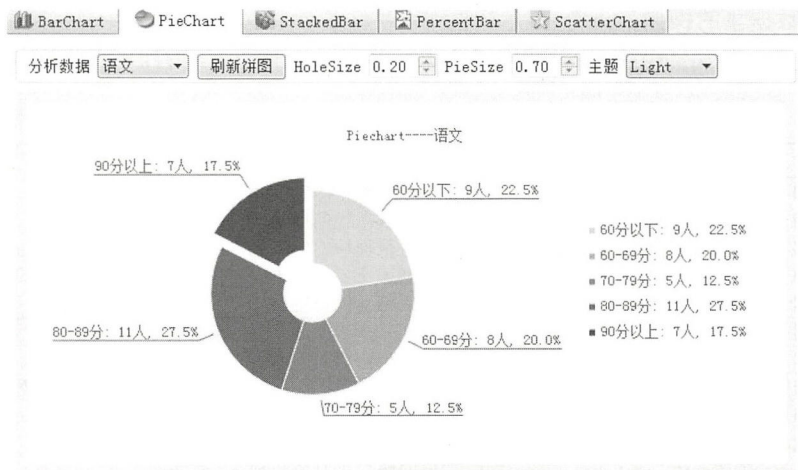


图 9-11 饼图绘制界面

界面上方的 HoleSize 可以设置饼图中心空心圆的大小, PieSize 可以设置饼图的大小,“主题”下拉列表框可以为图表选择不同的配色主题。

MainWindow 类中绘制饼图相关的函数有两个, iniPieChart()用于初始化,在 MainWindow 的构造函数里调用, buildPieChart()用于绘制饼图。

iniPieChart()函数用于创建 QChart 对象,并设置为 QChartView 类对象 chartViewPie 的显示图表,其代码与 iniBarChart()类似,不再赘述。

在图 9-11 的“分析数据”下拉列表框的 currentIndexChanged()信号的响应槽函数和“刷新饼图”按钮的响应代码里都调用 buildPieChart()函数绘制饼图。buildPieChart()函数的代码如下:

```
void MainWindow::buildPieChart()
{ //绘制饼图
    QChart *chart = ui->chartViewPie->chart(); //获取 chart 对象
    chart->removeAllSeries();
    int colNo=1+ui->cBoxCourse->currentIndex(); //获取分析对象
    QPieSeries *series = new QPieSeries(); //创建饼图序列
    series->setHoleSize(ui->spinHoleSize->value()); //饼图中间空心的大小
    for (int i=0;i<=4;i++) //添加分块数据
    {
        QtreeWidgetItem* item=ui->treeWidget->topLevelItem(i);
        series->append(item->text(0),item->text(colNo).toFloat());
    }

    QPieSlice *slice; //饼图分块
    for(int i=0;i<=4;i++) //设置每个分块的标签文字
    {
        slice =series->slices().at(i); //获取分块
        slice->setLabel(slice->label()+QString::asprintf(":%.0f 人, %.1f%%",
            slice->value(),slice->percentage()*100));
        connect(slice, SIGNAL(hovered(bool)),
            this, SLOT(on_PieSliceHighlight(bool)));
    }
    slice->setExploded(true); //最后一个设置为 exploded
    series->setLabelsVisible(true); //必须添加完 slice 之后再设置
    chart->addSeries(series); //添加饼图序列
    chart->setTitle("Piechart----"+ui->cBoxCourse->currentText());
    chart->legend()->setVisible(true); //图例
    chart->legend()->setAlignment(Qt::AlignRight);
}
```

用于绘制饼图的序列类是 QPieSeries,一般一个图表只有一个 QPieSeries 序列。

使用 QPieSeries::append()添加一个数据,实际上就是为 QPieSeries 增加了一个 QPieSlice 对象,就是饼图的一个分块。

QPieSeries::slices()返回已经添加的饼图分块,是一个 QPieSlice 类型的列表。获取某个分块 slice 后,用 slice->setLabel()函数设置其标签内容, slice->value()是数据块存储的数值, slice->percentage()是数据块在饼图中所占的百分比,是 0 到 1 之间的数。

另外,为每个分块对象 slice 的 hovered(bool)信号关联了一个槽函数 on\_PieSliceHighlight()。hovered(bool)信号在鼠标移到和移出对象上时发射,此槽函数用以实现某个数据块的动态弹出效果。

下面是 `on_PieSliceHighlight()` 槽函数的代码：

```
void MainWindow::on_PieSliceHighlight(bool show)
{ //鼠标移入、移出时发射 hovered() 信号，动态设置 exploded 效果
    QPieSlice* slice=(QPieSlice *)sender();
    slice->setExploded(show);
}
```

函数的输入参数 `show` 表示 `hovered()` 信号发射时，鼠标是移入还是移出。`show` 为 `true` 表示鼠标移入，否则为移出。代码首先通过 `sender()` 获取对象 `slice`，然后调用其 `setExploded()` 函数实现效果。若 `setExploded(true)`，则数据块在饼图中是弹出的，如图 9-11 中的“90 分以上”这个数据块就是处于弹出的状态；若 `setExploded(false)`，则数据块就是正常的状态。

**注意** 饼图没有坐标轴，所以无需设置坐标轴。

## 2. 饼图相关的主要类

饼图没有坐标轴，创建饼图主要涉及两个类。

- **QPieSeries**：饼图序列，一个图表一般只有一个饼图序列。
- **QPieSlice**：一个饼图的分块，一个饼图由多个分块组成。

**QPieSeries** 类是饼图序列，主要功能是对数据块的操作和饼图外观的设置，主要的函数功能见表 9-9（仅列出函数的返回数据类型，省略了输入参数）。

表 9-9 QPieSeries 类的主要函数功能

分组	函数	功能描述
分块操作	<code>bool append()</code>	添加一个分块到饼图
	<code>bool insert()</code>	在某个位置插入一个分块
	<code>bool remove()</code>	移除并删除一个分块
	<code>bool take()</code>	移除一个分块，但是并不删除数据块对象
	<code>void clear()</code>	清除序列所有的分块
	<code>QList&lt;QPieSlice *&gt; slices()</code>	返回序列的所有分块的列表
	<code>int count()</code>	返回序列分块的个数
	<code>bool isEmpty()</code>	如果序列是空的，返回 <code>true</code> ，否则返回 <code>false</code>
	<code>qreal sum()</code>	返回序列各分块的数值的和
外观	<code>void setHoleSize()</code>	设置饼图中心的空心圆的大小，在 0 至 1 之间
	<code>void setPieSize()</code>	设置饼图占图表矩形区的相对大小，0 是最小，1 是最大
	<code>void setLabelsVisible()</code>	设置分块的标签的可见性

**QPieSlice** 类是饼图上的一个分块，主要用于存储分块的数据，并决定分块的显示特性。**QPieSlice** 类的主要函数功能见表 9-10（仅列出函数的返回数据类型，省略了输入参数）。

表 9-10 QPieSlice 类的主要函数功能

分组	函数	功能描述
数据	<code>QPieSeries * series()</code>	返回分块所属的 <b>QPieSeries</b> 序列对象
	<code>void setValue()</code>	设置分块的数值，必须是正数
	<code>qreal percentage()</code>	返回本数据块的值在饼图中所有数据块的值的和中所占的百分比，数值在 0 到 1 之间。当饼图的数据块变化后自动更新
标签	<code>void setLabelVisible()</code>	设置标签的可见性
	<code>void setLabel()</code>	设置分块的标签文字

续表

分组	函数	功能描述
标签	void setLabelBrush()	设置标签的画刷
	void setLabelColor()	设置标签的颜色
	void setLabelFont()	设置标签的字体
	void setLabelPosition()	设置标签的位置, 输入参数是 QPieSlice::LabelPosition 枚举类型
外观	void setExploded()	设置分块是否弹出, 如果设置为 true, 分块具有弹出效果
	void setPen()	设置绘制分块的边框的画笔
	void setBorderColor()	设置边框的颜色, 是画笔颜色的便捷调用方式
	void setBorderWidth()	设置边框的线宽, 是画笔线宽的便捷调用方式
	void setBrush()	设置绘制分块的画刷
	void setColor()	设置分块的填充颜色, 是画刷颜色的便捷调用方式

### 9.3.5 堆叠柱状图

堆叠柱状图 (StackBar) 的绘制效果如图 9-12 所示, 它是柱状图的一种变体。图中有数学、语文、英语这 3 个数据集, 堆叠柱状图将 3 个数据集叠加成一个柱子来显示, 一个柱子中每个小段是一门课的分值, 柱子的高度表现了总分的大小。

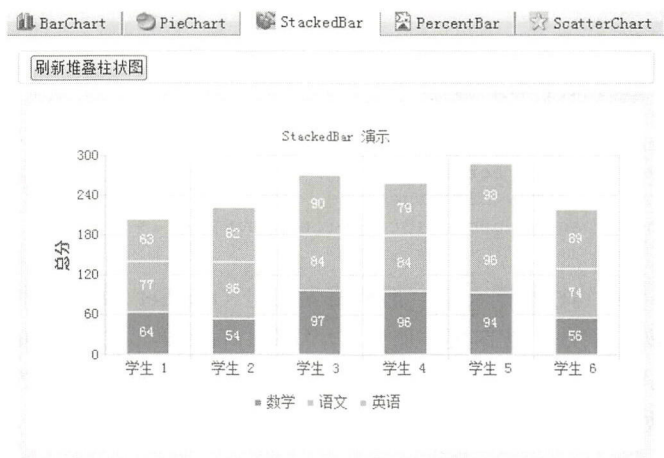


图 9-12 堆叠柱状图界面

iniStackedBar()用于图表的初始化, 其代码与 iniBarChart()相似, 这里不再赘述。

buildStackedBar()函数用于根据数据模型里的所有学生的 3 门课的分值绘制叠加柱状图, 实现代码如下:

```
void MainWindow::buildStackedBar()
{
    //绘制叠加柱状图
    QChart *chart = ui->chartViewStackedBar->chart(); //获取 QChart 对象
    chart->removeAllSeries();
    chart->removeAxis(chart->axisX());
    chart->removeAxis(chart->axisY());
    //创建三门课程的数据集
    QBarSet *setMath = new QBarSet(theModel->horizontalHeaderItem(colNoMath)->text());
    QBarSet *setChinese = new QBarSet(theModel->horizontalHeaderItem(colNoChinese)->text());
    QBarSet *setEnglish = new QBarSet(theModel->horizontalHeaderItem(colNoEnglish)->text());
}
```



```

for(int i=0;i<theModel->rowCount();i++)
{ //添加分数数据到数据集
    setMath->append(theModel->item(i,colNoMath)->text().toInt());
    setChinese->append(theModel->item(i,colNoChinese)->text().toInt());
    setEnglish->append(theModel->item(i,colNoEnglish)->text().toInt());
}
//创建 QStackedBarSeries 对象并添加数据集
QStackedBarSeries *series = new QStackedBarSeries();
series->append(setMath);
series->append(setChinese);
series->append(setEnglish);
series->setLabelsVisible(true); //显示每段的标签
chart->addSeries(series); //添加序列到图表
//创建横轴
QStringList categories;
for (int i=0;i<theModel->rowCount();i++)
    categories <<theModel->item(i,colNoName)->text();
QBarCategoryAxis *axisX = new QBarCategoryAxis(); //类别坐标轴，作为横轴
axisX->append(categories);
chart->setAxisX(axisX, series);
axisX->setRange(categories.at(0), categories.at(categories.count()-1));
//数值坐标轴，作为纵轴
QValueAxis *axisY = new QValueAxis; //数值坐标轴，作为纵轴
axisY->setRange(0, 300);
axisY->setTitleText("总分");
axisY->setTickCount(6);
axisY->setLabelFormat("%.0f"); //标签格式
chart->setAxisY(axisY, series);
chart->legend()->setVisible(true);
chart->legend()->setAlignment(Qt::AlignBottom);
}

```

创建堆叠柱状图与创建柱状图类似，都需要先创建 `QBarSet` 数据集并添加数据，然后创建 `QStackedBarSeries` 序列，将 3 个数据集添加到这个序列。横轴采用 `QBarCategoryAxis` 类型的坐标轴，纵轴采用 `QValueAxis` 类型的坐标轴。除了使用 `QStackedBarSeries` 类作为序列，其他用到的类与 `buildBarChart()` 函数里的都相同。

### 9.3.6 百分比柱状图

绘制百分比柱状图的界面如图 9-13 所示。图中有数学、语文、英语 3 个数据集，3 个数据集的数据来源是按分数统计的结果，3 个数据集叠加显示，显示效果类似于叠加柱状图，但是每个柱子的 3 个数据的和是相同的，等于学生总人数，所以，所有的柱子是等高度的。显示标签时，显示的是每段所占的百分比。

为避免某个分数段统计人数为 0，将学生人数设置为 40 人，就是修改 `mianwindow.h` 文件中定义的符号 `iniDataRowCount` 为 40。

`iniPercentBar()` 用于图表的初始化，与 `iniBarChart()` 代码相似，这里不再赘述。

`buildPercentBar()` 函数用于根据统计数据绘制百分比柱状图，其代码如下：

```

void MainWindow::buildPercentBar()
{ //绘制百分比柱状图

```

```

QChart *chart = ui->chartViewPercentBar->chart();
chart->removeAllSeries();
chart->removeAxis(chart->axisX());
chart->removeAxis(chart->axisY());
//创建数据集
QBarSet *setMath = new QBarSet(theModel->horizontalHeaderItem(colNoMath)->text());
QBarSet *setChinese = new QBarSet(theModel->horizontalHeaderItem(colNoChinese)->text());
QBarSet *setEnglish= new QBarSet(theModel->horizontalHeaderItem(colNoEnglish)->text());
QTreeWidgetItem *item; //节点
QStringList categories;
for (int i=0;i<=4;i++)
{ //从分数段统计数据表里获取数据, 添加到数据集
    item=ui->treeWidget->topLevelItem(i);
    categories<<item->text(0); //用作横坐标的标签
    setMath->append(item->text(colNoMath).toFloat());
    setChinese->append(item->text(colNoChinese).toFloat());
    setEnglish->append(item->text(colNoEnglish).toFloat());
}
QPercentBarSeries *series = new QPercentBarSeries(); //序列
series->append(setMath);
series->append(setChinese);
series->append(setEnglish);
series->setLabelsVisible(true); //显示百分比
chart->addSeries(series);

QBarCategoryAxis *axisX = new QBarCategoryAxis(); //横坐标
axisX->append(categories);
chart->setAxisX(axisX, series);
axisX->setRange(categories.at(0), categories.at(categories.count()-1));
QValueAxis *axisY = new QValueAxis; //纵坐标
axisY->setRange(0, 100);
axisY->setTitleText("百分比");
axisY->setTickCount(6);
axisY->setLabelFormat("%.1f"); //标签格式
chart->setAxisY(axisY, series);
chart->legend()->setVisible(true);
chart->legend()->setAlignment(Qt::AlignBottom);
}

```

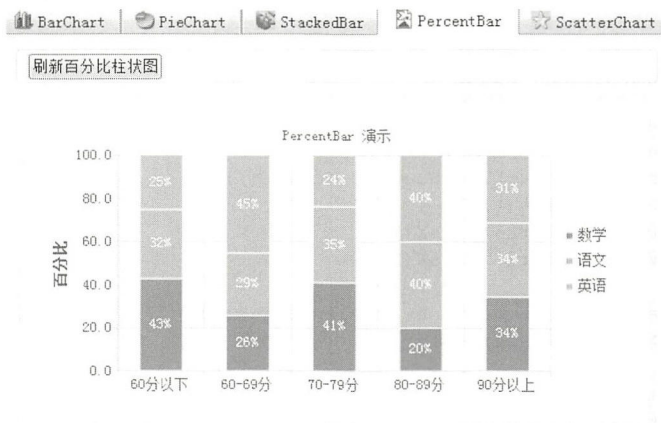


图 9-13 百分比柱状图界面

绘制百分比柱状图也需要先创建 `QBarSet` 数据集，这里创建了 3 个数据集，从统计数据表里分别获取 3 门课各个分数段的人数。

绘制百分比柱状图使用 `QPercentBarSeries` 类的序列，添加数据集，设置  $X$  和  $Y$  轴的坐标轴即可。若显示数据标签，显示的是某个分数段内某门课的人数的百分比。

### 9.3.7 散点图和光滑曲线图

在窗口绘图区的“ScatterChart”页面可以绘制散点图和光滑曲线图，绘图效果如图 9-14 所示。

绘制散点图使用 `QScatterSeries` 序列类。它在添加数据点后，将数据点以散点的形式显示，如图 9-14 中的“散点”序列。可以控制散点的形状、大小和颜色等属性。

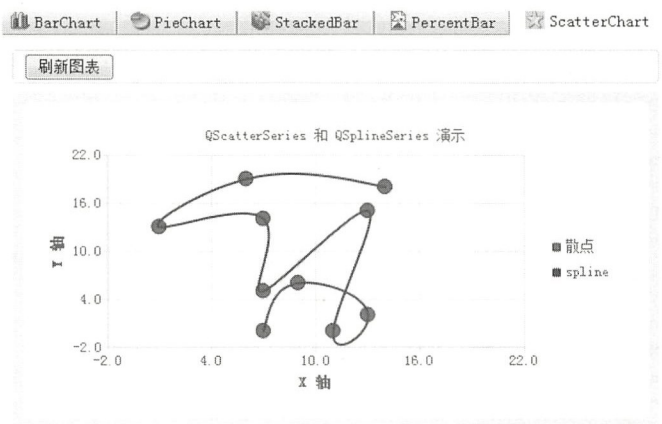


图 9-14 散点与 spline 图

绘制光滑曲线使用 `QSplineSeries` 序列类，它在两点之间连线时采用差值算法，从而形成光滑的曲线，而绘制折线图采用的是 `QLineSeries` 序列类，两点间只是简单的用直线连接。

实例中绘制散点和光滑曲线图使用随机数据，与界面左侧的学生分数数据无关。

`iniScatterChart()`用于图表的初始化，与 `iniBarChart()`的代码相似，这里不再赘述。

单击绘图页面上方的“刷新图表”按钮，将调用 `buildScatterChart()`函数绘制散点图和 spline 曲线，`buildScatterChart()`函数的代码如下：

```
void MainWindow::buildScatterChart()
{ //绘制 QScatterSeries 和 QSplineSeries 图
    QChart *chart = ui->chartViewScatter->chart();
    chart->removeAllSeries();
    chart->removeAxis(chart->axisX());
    chart->removeAxis(chart->axisY());

    QSplineSeries *seriesLine = new QSplineSeries(); //光滑曲线序列
    seriesLine->setName("spline");
    QPen pen;
    pen.setColor(Qt::blue);
    pen.setWidth(2);
    seriesLine->setPen(pen);
```

```

QScatterSeries *series0 = new QScatterSeries(); //散点序列
series0->setName("散点");
series0->setMarkerShape(QScatterSeries::MarkerShapeCircle);
series0->setBorderColor(Qt::black);
series0->setBrush(QBrush(Qt::red));
series0->setMarkerSize(12);

qrand(QTime::currentTime().second()); //随机数种子
for (int i=0;i<10;i++)
{
    int x=(qrand() % 20); //0 到 20 之间的随机数
    int y=(qrand() % 20);
    series0->append(x,y); //散点序列
    seriesLine->append(x,y); //光滑曲线序列
}
chart->addSeries(series0);
chart->addSeries(seriesLine);

chart->createDefaultAxes(); //创建缺省的坐标轴
chart->axisX()->setTitleText("X 轴");
chart->axisX()->setRange(-2,22);
chart->axisY()->setTitleText("Y 轴");
chart->axisY()->setRange(-2,22);
chart->legend()->setVisible(true);
chart->legend()->setAlignment(Qt::AlignRight);
}

```

函数创建了一个 QSplineSeries 序列对象 seriesLine, 一个 QScatterSeries 序列对象 series0。

散点序列的数据点形状采用函数 setMarkerShape()进行设置, 传递的参数是一个 QScatterSeries::MarkerShape 枚举类型的值, 这个枚举类型只有两种取值:

- QScatterSeries::MarkerShapeCircle, 圆形;
- QScatterSeries::MarkerShapeRectangle, 矩形。

setBorderColor()函数用于设置数据点形状边框的颜色, setBrush()函数可以设置填充颜色, setMarkerSize()函数设置数据点形状的大小。

两个序列创建后, 采用随机函数生成数据点, 两个序列添加相同的数据点。

创建坐标轴时采用了 QChart::createDefaultAxes()函数生成缺省的坐标轴, 创建缺省的坐标轴之后, 还是可以通过 QChart::axisX()或 QChart::axisY()获取图表的 X 坐标轴或 Y 坐标轴进行坐标轴的属性设置, 例如设置坐标轴标题和数值范围。

## 9.4 图表的其他操作

### 9.4.1 实例功能概述

前面介绍了 QChart 绘图的基本功能, 以及几种常见图表的绘制方法, 本节介绍 QChart 绘图的一些高级的用法, 也是经常需要实现的一些功能。实例 samp9\_4 用于演示这些功能的实现, 图



9-15 是实例运行界面。

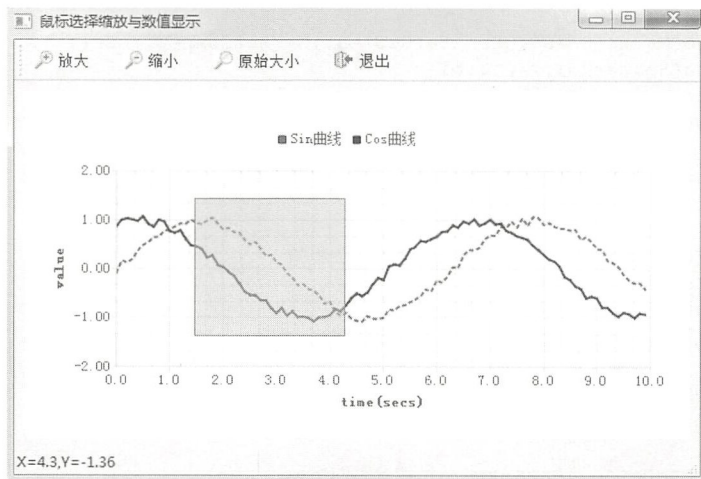


图 9-15 实例 samp9\_4 运行界面

该实例具有如下功能：

- 鼠标在图表上移动时，在状态栏里显示鼠标当前位置的坐标数值，这个功能在一般的数据曲线显示图表中是经常需要实现的；
- 图例具有类似于 QCheckBox 的功能，单击图例的图标可以显示或隐藏对应的序列；
- 通过工具栏按钮可以进行图表缩放操作；
- 鼠标在图表上可以选择矩形区域进行放大，单击鼠标右键则恢复图表大小；
- 使用按键可以实现图表的快捷操作，如“+”键进行放大、“-”键进行缩小、Home 键恢复原始大小，使用上、下、左、右光标键可以移动序列的显示位置等。

## 9.4.2 自定义 QWChartView 类

QChart 和 QChartView 是基于 Graphics View 结构的绘图类，要对一个 QChart 图表进行鼠标和按键操作，需要在 QChartView 组件里对鼠标和按键事件进行处理，这就需要自定义一个从 QChartView 继承的类，这与实例 samp8\_5 中从 QGraphicsView 继承一个自定义图形视图类，实现鼠标和按键操作的原理类似。

自定义一个 QWChartView 类，它从 QChartView 继承而来，对鼠标和按键事件进行处理。QWChartView 类的定义如下：

```
class QWChartView : public QChartView
{
    Q_OBJECT
private:
    QPoint beginPoint; //选择矩形区的起点
    QPoint endPoint;   //选择矩形区的终点
protected:
    void mousePressEvent(QMouseEvent *event); //鼠标左键按下
```

```

void mouseMoveEvent(QMouseEvent *event); //鼠标移动
void mouseReleaseEvent(QMouseEvent *event); //鼠标释放左键
void keyPressEvent(QKeyEvent *event); //按键事件
public:
    explicit QWChartView(QWidget *parent = 0);
    ~QWChartView();
signals:
    void mouseMovePoint(QPoint point); //鼠标移动信号
};

```

QWChartView 类中定义了两个私有变量 beginPoint 和 endPoint 用于鼠标框选矩形区域的起点和终点。QWChartView 重定义了鼠标的 3 个事件函数和键盘按键事件的函数，定义了一个信号 mouseMovePoint(QPoint point)，在 mouseMoveEvent() 事件里发射此信号并传递鼠标光标处的屏幕坐标，用于在主窗口里实现鼠标在图表上移动时显示当前位置的坐标。

下面是 QWChartView 类各个函数的实现。

```

QWChartView::QWChartView(QWidget *parent):QChartView(parent)
{ //构造函数
    this->setDragMode(QGraphicsView::RubberBandDrag);
    this->setMouseTracking(true); //必须开启此功能，缺省为 false
}
void QWChartView::mousePressEvent(QMouseEvent *event)
{ //鼠标左键按下，记录 beginPoint
    if (event->button() == Qt::LeftButton)
        beginPoint = event->pos();
    QChartView::mousePressEvent(event);
}
void QWChartView::mouseMoveEvent(QMouseEvent *event)
{ //鼠标移动事件
    QPoint point = event->pos();
    emit mouseMovePoint(point);
    QChartView::mouseMoveEvent(event);
}
void QWChartView::mouseReleaseEvent(QMouseEvent *event)
{ //鼠标左键释放事件
    if (event->button() == Qt::LeftButton)
    { //鼠标左键释放，获取矩形框的 endPoint，进行缩放
        endPoint = event->pos();
        QRectF rectF;
        rectF.setTopLeft(this->beginPoint);
        rectF.setBottomRight(this->endPoint);
        this->chart()->zoomIn(rectF);
    }
    else if (event->button() == Qt::RightButton)
        this->chart()->zoomReset(); //鼠标右键释放，resetZoom
    QChartView::mouseReleaseEvent(event);
}
void QWChartView::keyPressEvent(QKeyEvent *event)
{ //按键控制
    switch (event->key()) {
        case Qt::Key_Plus:
            chart()->zoom(1.2); break;
        case Qt::Key_Minus:
            chart()->zoom(0.8); break;
    }
}

```

```

case Qt::Key_Left:
    chart()->scroll(10, 0);        break;
case Qt::Key_Right:
    chart()->scroll(-10, 0);       break;
case Qt::Key_Up:
    chart()->scroll(0, -10);       break;
case Qt::Key_Down:
    chart()->scroll(0, 10);        break;
case Qt::Key_PageUp:
    chart()->scroll(0, 50);        break;
case Qt::Key_PageDown:
    chart()->scroll(0, -50);       break;
case Qt::Key_Home:
    chart()->zoomReset();          break;
default:
    QGraphicsView::keyPressEvent(event);
}
}

```

在构造函数里，`setDragMode(QGraphicsView::RubberBandDrag)`将视图组件鼠标拖动选择方式设置为“橡皮框”形式，即鼠标左键按下时，随着鼠标拖动显示一个矩形选择框。

`setMouseTracking(true)`将鼠标跟踪设置为 `true`（缺省为 `false`）。如果不设置为 `true`，窗口组件只在某个鼠标按键按下时才接收鼠标移动事件，设置为 `true` 之后，只要鼠标移动就会发射 `mouseMoveEvent()` 事件。

`mousePressEvent(QMouseEvent *event)`是在鼠标左键或右键按下时发生的事件，在响应程序里先判断鼠标左键是否按下，如果是鼠标左键按下了，就用变量 `beginPoint` 记录鼠标在视图组件中的位置。

`mouseReleaseEvent(QMouseEvent *event)`是在鼠标左键或右键释放时发生的事件，若是鼠标左键释放，则用变量 `endPoint` 记录鼠标位置坐标，`beginPoint` 和 `endPoint` 就定义了鼠标框选的矩形区域，用关联的 `chart` 组件的 `zoomIn()` 函数显示这个矩形区域实现放大。

`mouseMoveEvent(QMouseEvent *event)`是鼠标在图表上移动时发生的事件，通过 `event->pos()` 获取鼠标在视图组件中的坐标 `point`，然后发射信号 `mouseMovePoint(point)`。在使用 `QWChartView` 类组件的主窗口里，可以定义槽函数对此信号作出响应，通过传递的参数将视图坐标转换为图表的坐标，从而实现鼠标位置的数值显示。

`keyPressEvent(QKeyEvent *event)`是键盘按键按下时发生的事件，从 `event->key()` 可以获得按下按键的名称，判断按键，然后做出缩放、移动等操作。

`QChart::zoom(qreal factor)`函数对图表显示区的内容进行缩放，`factor` 大于 1 是放大，小于 1 是缩小，缩放后坐标轴的范围会自动变化。

`QChart::scroll(qreal dx, qreal dy)`函数将图表内容在平面上平移，平移后坐标轴会自动变化。

`QChart::zoomReset()`函数将取消所有缩放变化，恢复原始的大小。

### 9.4.3 主窗口类的设计

创建一个 `Widget Application` 项目，主窗口继承自 `QMainWindow`，按照图 9-15 进行窗口可视

化设计。主窗口上放置一个 QGraphicsView 组件后提升为 QWChartView 类。

主窗口类 MainWindow 的定义如下：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QLabel *labXYValue; //状态栏显示文字标签
    void createChart(); //创建图表
    void prepareData(); //准备数据
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    void on_LegendMarkerClicked(); //图例单击槽函数, 自定义槽函数
    void on_mouseMovePoint(QPoint point); //鼠标移动事件, 自定义槽函数

    void on_actZoomReset_triggered(); //工具栏按钮, 原始大小
    void on_actZoomIn_triggered(); //工具栏按钮, 放大
    void on_actZoomOut_triggered(); //工具栏按钮, 缩小
private:
    Ui::MainWindow *ui;
};
```

QWChartView 类已经实现了鼠标拖动区域放大功能和按键快捷操作功能, 在主窗口类 MainWindow 中无需再实现这些功能。主窗口需要实现的功能包括:

- 绘制图表, createChart()函数用于初始化图表组件, prepareData()函数用于为图表创建序列并添加数据。
- 定义一个槽函数 on\_mouseMovePoint(QPoint), 与 QWChartView 的 mouseMovePoint(QPoint) 信号相关联, 用于实时显示鼠标光标处的坐标值。
- 定义一个槽函数 on\_LegendMarkerClicked(), 与图表的图例 marker 的 clicked()信号相关联, 实现单击图例时显示或隐藏相关序列。

#### 9.4.4 实时显示光标处的数值

主窗口类 MainWindow 的构造函数实现如下:

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setUpUi(this);
    this->setCentralWidget(ui->chartView);
    labXYValue = new QLabel("X=, Y= "); //状态栏显示鼠标点的坐标
    labXYValue->setMinimumWidth(200);
    ui->statusBar->addWidget(labXYValue);
    createChart(); //创建图表
    prepareData(); //生成数据
    QObject::connect(ui->chartView, SIGNAL(mouseMovePoint(QPoint)),
        this, SLOT(on_mouseMovePoint(QPoint))); //鼠标移动事件
}
```



构造函数调用 `createChart()` 创建图表，调用 `prepareData()` 为图表添加数据。最后，将窗口上的 `chartView` 组件的 `mouseMovePoint(QPoint)` 信号与自定义槽函数 `on_mouseMovePoint(QPoint)` 关联起来，用于实现鼠标在图表上移动时在状态栏里显示光标处的数值。

`createChart()` 和 `prepareData()` 的代码不详细介绍了，它们的作用就是创建两个 `QLineSeries` 序列，显示正弦、余弦曲线，其实现与实例 `samp9_1` 类似。

槽函数 `on_mouseMovePoint(QPoint)` 的实现代码如下：

```
void MainWindow::on_mouseMovePoint(QPoint point)
{ // 鼠标移动响应
    QPointF pt=ui->chartView->chart()->mapToValue(point); // 转换为图表的数值
    labXYValue->setText(QString::asprintf("X=%.1f,Y=%.2f",pt.x(),pt.y()));
}
```

传递的参数 `point` 是 `chartView` 的坐标，用 `QChart::mapToValue()` 将视图的坐标转换为图表的坐标，这与 `Graphics View` 里视图坐标到场景坐标的转换类似。转换后得到的坐标 `pt` 是图表的数值坐标，即显示区在两个坐标轴定义下的坐标，在状态栏上显示此数值坐标的内容，就实现了实时显示鼠标光标处的数值的功能。

## 9.4.5 QLegendMarker 的使用

图表创建序列后会自动创建图例，`QLegend` 有一个只读属性 `markers`，是一个列表类型，存储了每个序列的 `QLegendMarker` 对象。`markers` 函数的原型如下：

```
QList<QLegendMarker *> QLegend::markers(QAbstractSeries *series = Q_NULLPTR)
```

如果不指定序列，则返回图表所有序列的 `LegendMarker` 对象列表。

`QLegendMarker` 是直接继承自 `QObject` 的类，用于在图例中对关联的序列进行操作。`QLegendMarker` 有两个部分，一个反映了序列颜色的颜色框，一个是反映序列名称的标签。`QLegendMarker` 类的主要函数见表 9-11（省略了函数参数中的 `const` 关键字）。

表 9-11 QLegendMarker 类的主要函数

函数原型	功能描述
<code>void setVisible(bool visible)</code>	设置图例标记的可见性
<code>void setLabel(QString &amp;label)</code>	设置标签，即图例中的序列的名称
<code>void setFont(QFont &amp;font)</code>	设置标签的字体
<code>QAbstractSeries * series()</code>	返回关联的序列
<code>LegendMarkerType type()</code>	返回图例标记的类型，取决于关联的序列的类型，有： <code>QLegendMarker::LegendMarkerTypeArea</code> <code>QLegendMarker::LegendMarkerTypeBar</code> <code>QLegendMarker::LegendMarkerTypePie</code> <code>QLegendMarker::LegendMarkerTypeXY</code> <code>QLegendMarker::LegendMarkerTypeBoxPlot</code> <code>LegendMarkerTypeXY</code> 用于所有从 <code>QXYSeries</code> 继承的序列类

`QLegendMarker::visible` 属性控制图例标记的可见性，如果设置为 `false`，在图例中就不会显示这个图例标记。可以将 `QLegendMarker` 当作一个 `QCheckBox` 使用，单击图例中相应的 `LegendMarker` 时，显示或隐藏相关联的序列，这只需要设计一个槽函数响应 `QLegendMarker` 的 `clicked()` 信号即可。

在 `createChart()` 函数中为图表创建两个 `QLineSeries` 序列后，在最后添加如下代码：

```
foreach (QLegendMarker* marker, chart->legend()->markers())
    connect(marker, SIGNAL(clicked()),
            this, SLOT(on_LegendMarkerClicked()));
```

该语句通过图例的 `markers()` 获取其 `QLegendMarker` 列表，然后为每个 `marker` 的 `clicked()` 信号关联自定义的槽函数 `on_LegendMarkerClicked()`。

下面是槽函数 `on_LegendMarkerClicked()` 的代码：

```
void MainWindow::on_LegendMarkerClicked()
{ //单击图例的 marker 的响应
    QLegendMarker* marker = qobject_cast<QLegendMarker*> (sender());
    switch (marker->type()) //marker 的类型
    {
        case QLegendMarker::LegendMarkerTypeXY: //QLineSeries 序列
        {
            marker->series()->setVisible(!marker->series()->isVisible());
            marker->setVisible(true);
        }
    }
}
```

这里，首先将产生信号的对象 `sender()` 转换为 `QLegendMarker` 类对象 `marker`，然后根据 `marker->type()` 判断图例标记的类型，本实例里创建的是 `QLineSeries` 序列，所以图例标记的类型是 `QLegendMarker::LegendMarkerTypeXY`。

通过 `marker->series()` 获取关联的序列，然后交替设置其可见性，即：

```
marker->series()->setVisible(!marker->series()->isVisible());
```

`marker` 的可见性总是设置为 `true`，因为若设置为 `false`，图例标记也会被隐藏，就无法再用鼠标单击操作了。

## 9.4.6 图表的缩放

在 `QWChartView` 类里，对鼠标事件进行了处理，用鼠标拖动一个矩形框时，图表放大显示所选的矩形区域，而且可以连续多次放大，单击鼠标右键时，又恢复原始大小。

在 `QWChartView` 类里还定义了键盘按键操作，运行时按相应的按键就可以进行放大、缩小、移动等操作。

主窗口工具栏上的“放大”“缩小”“原始大小”按钮可以实现图表的缩放操作，3 个按钮的响应代码如下：

```
void MainWindow::on_actZoomReset_triggered()
{
    ui->chartView->chart()->zoomReset(); //恢复原始大小
}
void MainWindow::on_actZoomIn_triggered()
{
    ui->chartView->chart()->zoom(1.2); //放大
}
void MainWindow::on_actZoomOut_triggered()
{
    ui->chartView->chart()->zoom(0.8); //缩小
}
```

# Data Visualization

Data Visualization 是 Qt 提供的用于数据三维显示的模块。在 Qt 5.7 以前只有商业版才有此模块，而从 Qt 5.7 开始此模块在社区版本里也可以免费使用了。Data Visualization 用于数据的三维显示，包括三维柱状图、三维空间散点、三维曲面等。Data Visualization 与 Qt Charts 类似，也是基于 Qt 的图形视图架构。Data Visualization 的功能无法和一些专业的三维显示类库相提并论，但是对于一些简单的三维数据显示是比较实用的，例如一些科学计算结果的三维显示。

本章介绍 Data Visualization 模块的一些主要功能的使用，主要是三维柱状图、三维空间散点和三维曲面的显示。

## 10.1 Data Visualization 模块概述

Data Visualization 的三维显示功能主要由 3 种三维图形类来实现，分别是三维柱状图类 Q3DBars，三维空间散点类 Q3DScatter，三维曲面类 Q3DSurface。这 3 个类的父类是 QAbstract3DGraph，是从 QWindow 继承而来的，继承关系如图 10-1 所示。

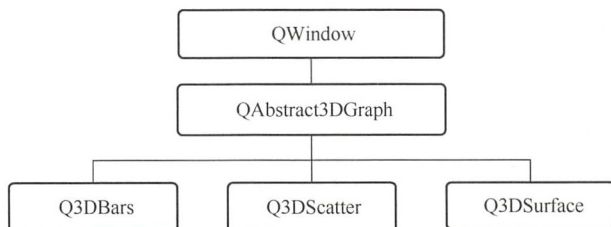


图 10-1 三种 Data Visualization 图形类的继承关系

Data Visualization 与 Qt Charts 类似，都基于 Qt 的图形视图结构，所以一个三维图形也是由图表、序列、坐标轴等元素组成的。Q3DBars、Q3DScatter、Q3DSurface 相当于 Qt Charts 中的 QChart，而每一种三维图形对应一种三维序列，Data Visualization 中的 3 种序列类见图 10-2。

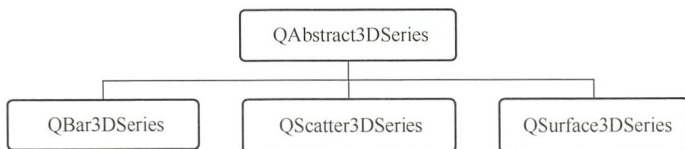


图 10-2 三种三维图形序列类

一种序列类只能用于某种三维图形类,如 `QBar3DSeries` 只能用作三维柱状图 `Q3DBars` 的序列,而不能作为三维散点图 `Q3DScatter` 的序列。在一个图中可以有多个同类型的序列,如三维曲面图 `Q3DSurface` 中可以有多个 `QSurface3DSeries` 序列,用于显示不同的曲面。

与 `QChart` 有坐标轴类一样,三维图形也有坐标轴类。有两种三维坐标轴类, `QValue3DAxis` 用于数值型坐标轴, `QCategory3DAxis` 用于文字型坐标轴,它们都继承自 `QAbstract3DAxis` (如图 10-3 所示)。

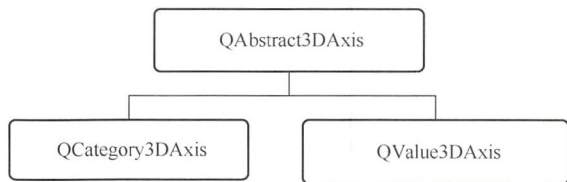


图 10-3 两种用于三维图形的坐标轴

`Data Visualization` 模块中有数据代理 (data proxy) 类,数据代理类就是与序列对应,用于存储序列的数据的类。因为三维图形类型不一样,存储数据的结构也不一样,例如三维散点序列 `QScatter3DSeries` 存储的是一些三维数据点的坐标,只需要用一维数组或列表就可以存储这些数据,而 `QSurface3DSeries` 序列存储的数据点在水平面上是均匀网格分布的,需要二维数组才可以存储相应的数据。为此,对于每一种序列,都有一个数据代理类,它们都继承自 `QAbstractDataProxy`,每个数据代理类还有一个基于项数据模型的数据代理子类 (如图 10-4 所示)。

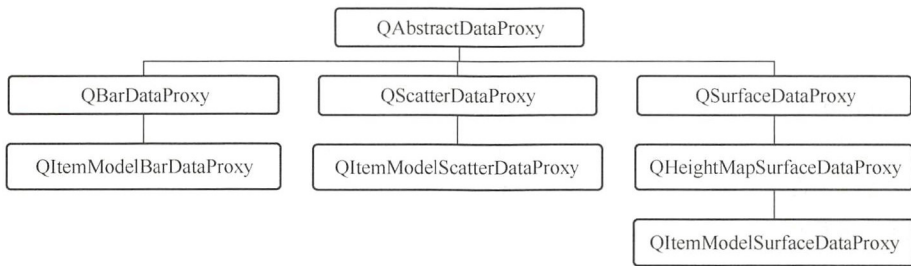


图 10-4 数据代理类的集成关系

对于三维曲面序列 `QSurface3DSeries`,还有一个专门用于显示地图高程数据的数据代理类 `QHeightMapSurfaceDataProxy`,可以将一个图片表示的高程数据显示为三维曲面。用户也可以根据需要从 `QAbstractDataProxy` 继承,定义自己的数据代理类。

要在项目中使用 `Data Visualization` 模块,需要在项目配置文件中添加下面一行语句:

```
Qt += datavisualization
```

在使用 `Data Visualization` 模块中的类的头文件或源程序文件中,还需要加入下面两行语句。如果只需使用模块中的部分类,可以单独包含某些类。

```
#include <QtDataVisualization>
using namespace QtDataVisualization;
```

## 10.2 三维柱状图

### 10.2.1 实例功能

通过一个三维柱状图的实例来说明使用 `Data Visualization` 的类绘制三维图形的基本原理。实



例 samp10\_1 使用 Q3DBars 绘制一个三维柱状图，并在界面上对其一些常见属性和操作进行控制，程序运行效果如图 10-5 所示。

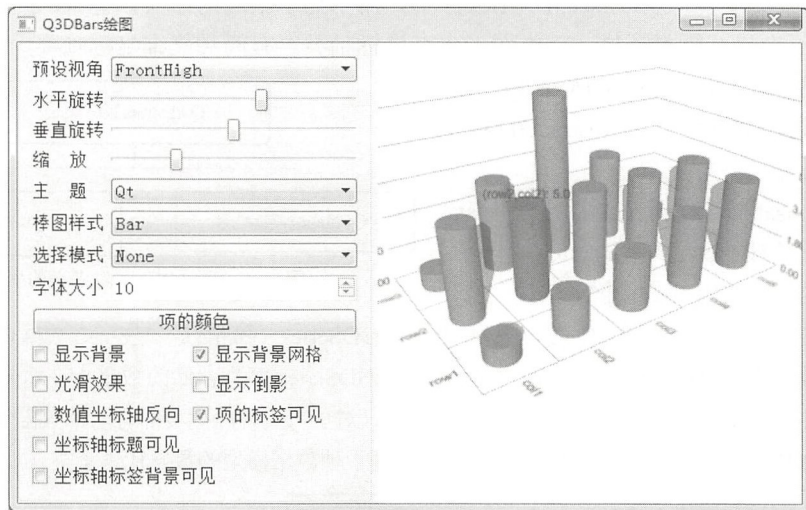


图 10-5 三维柱状图绘图

实例 samp10\_1 运行时，右侧是用 Q3DBars 绘制的三维柱状图；左侧是对三维柱状图的一些操作和属性设置，包括预设视角的选择，水平、垂直方向的旋转以及缩放，属性设置包括图表主题、选择方式、棒图项的样式，以及一些显示效果的设置。

无需额外编程或设置，程序运行时在图表单击左键可以选择图表中的某个项、某个列或行（与选择方式的设置有关），按住鼠标右键时上下、左右移动可以进行水平和垂直方向的旋转，鼠标滚轮可以进行缩放。

## 10.2.2 主窗口设计

实例 samp10\_1 是主窗口继承自 QMainWindow 的应用程序，可视化创建主窗体。

由于三维图形类 Q3DBars、Q3DScatter 和 Q3DSurface 都是从 QWindow 继承而来的（见图 10-1），不能简单使用 QWidget 组件作为 Q3DBars 组件的容器，也就是不能在主窗口上放置一个 QWidget 组件，然后作为 Q3DBars 组件的容器，而是需要使用 QWidget::createWindowContainer() 动态创建 QWidget 作为 Q3DBars 的容器。

在 UI 设计器中设计主窗体的界面如图 10-6 所示，只设计用于三维图表属性控制的界面组件，用一个 QGroupBox 组件容纳所有这些三维属性控制界面组件，将这个 QGroupBox 组件命名为 groupBox。为 groupBox 内的组件设计好布局，删除主窗口的工具栏、菜单栏和状态栏。窗口右侧区域是空白，是为动态创建 Q3DBars 的 QWidget 容器预留的空间。

主窗口类的头文件 mainwindow.h 中的定义如下：

```
#include <QMainWindow>
#include <QtDataVisualization>
```



```

        this->setCentralWidget(splitter);
    }

```

构造函数里首先调用 `iniGraph3D()` 创建三维图表及其容器 `graphContainer`，然后创建一个分隔条组件 `splitter`，将可视化设计的控制面板 `groupBox` 与动态创建的图表容器 `graphContainer` 添加到分隔条里，作为主窗口的中心组件。

这样在运行时就获得图 10-1 的界面，界面具体自动填充效果，并且可以分割左侧的控制面板和右侧的图表。

`iniGraph3D()` 函数创建图表，其实现代码如下：

```

void MainWindow::iniGraph3D()
{
    //创建图表
    graph3D = new Q3DBars();
    graphContainer = QWidget::createWindowContainer(graph3D);
    //创建坐标轴
    QStringList rowLabs;
    QStringList colLabs;
    rowLabs << "row1" << "row2" << "row3";
    colLabs << "col1" << "col2" << "col3" << "col4" << "col5";
    QValue3DAxis *axisV=new QValue3DAxis; //数值坐标轴
    axisV->setTitle("value");
    axisV->setTitleVisible(true);
    QCategory3DAxis *axisRow=new QCategory3DAxis; //行坐标轴
    axisRow->setTitle("row axis");
    axisRow->setLabels(rowLabs);
    axisRow->setTitleVisible(true);
    QCategory3DAxis *axisCol=new QCategory3DAxis; //列坐标轴
    axisCol->setTitle("column axis");
    axisCol->setLabels(colLabs);
    axisCol->setTitleVisible(true);
    graph3D->setValueAxis(axisV);
    graph3D->setRowAxis(axisRow);
    graph3D->setColumnAxis(axisCol);
    //创建序列
    series = new QBar3DSeries;
    series->setMesh(QAbstract3DSeries::MeshCylinder); //棒图形状
    series->setItemLabelFormat("@rowLabel,@colLabel: %.1f"); //标签格式
    graph3D->addSeries(series);
    //添加数据
    QBarDataArray *dataSet = new QBarDataArray; //数据数组
    dataSet->reserve(rowLabs.count());
    QBarDataRow *dataRow = new QBarDataRow;
    *dataRow << 1 << 2 << 3 << 4 << 5; //第1行数据，有5列
    dataSet->append(dataRow);
    QBarDataRow *dataRow2 = new QBarDataRow;
    *dataRow2 << 5 << 5 << 5 << 5 << 5; //第2行数据，有5列
    dataSet->append(dataRow2);
    QBarDataRow *dataRow3 = new QBarDataRow;
    *dataRow3 << 1 << 5 << 9 << 5 << 1; //第3行数据，有5列
    dataSet->append(dataRow3);

    series->dataProxy()->resetArray(dataSet);
}

```

程序首先创建 Q3DBars 类的实例 graph3D，然后创建一个 QWidget 实例 graphContainer 作为其容器。创建 graphContainer 时使用了 QWidget 的静态函数 createWindowContainer()，这样创建的 QWidget 实例才可以作为 QWindow 及其子类组件的容器。

三维柱状图的水平面的两个轴采用 QCategory3DAxis 类型的坐标轴，是文字型标签坐标轴，分别为其设置轴标题和轴标签。垂直方向表示项的数值大小，所以使用 QValue3DAxis 类型的坐标轴。创建坐标轴后，分别设置为柱状图行坐标轴、列坐标轴和数值坐标轴，即：

```
graph3D->setValueAxis(axisV);
graph3D->setRowAxis(axisRow);
graph3D->setColumnAxis(axisCol);
```

与 Q3DBars 对应使用的序列是 QBar3DSeries 类，创建序列实例对象 series，设置棒图性质和标签格式后添加到图表。

然后创建 QBarDataRow 数组 dataSet，其元素个数等于三维柱状图的行数，也就是 dataSet 的每个元素是一个行的数据。

一个行的数据是 QBarDataRow 类型，例如，创建第 1 行的数据并添加到柱状图的数据数组里的代码如下：

```
QBarDataRow *dataRow = new QBarDataRow;
*dataRow << 1 << 2<< 3<< 4<<5; //第 1 行数据，有 5 列
dataSet->append(dataRow);
```

实际上，QBarDataRow 和 QBarDataRow 都不是类，而是类型定义，在 Qt 的源代码中，它们的定义如下：

```
typedef QVector<QBarDataItem> QBarDataRow;
typedef QList<QBarDataRow *> QBarDataRowArray;
```

所以，QBarDataRow 是 QBarDataItem 的向量，QBarDataRow 是 QBarDataRow 的列表。数据的基本元素是 QBarDataItem，也就是柱状图中的一个项，QBarDataItem 只有 value 和 rotation 两个属性，用于存储项的数值和旋转角度。

在图 10-5 中，柱状图的数据有 3 行 5 列，添加数据时，每次添加一行的数据，但是属于一个序列。

设置完数据后，为序列的数据代理设置数据，语句如下：

```
series->dataProxy()->resetArray(dataSet);
```

程序里没有为序列先创建 QBarDataProxy 类型的数据代理，然后再添加到序列里。调用 series->dataProxy()使用内部缺省的数据代理，resetArray(dataSet)将清除数据代理原有的数据，并使用 dataSet 作为数据源。

## 10.2.4 三维柱状图属性设置

图 10-5 左侧的控制面板用于图表的操作和一些常用属性的设置。

### 1. 预设视角

3 种三维图表的父类 QAbstract3DGraph 有一个函数 scene()可以获得图表的场景，是一个



Q3DScene 类。Q3DScene 是三维图的场景，包含一个相机（camera）和一个光源，还提供一个 3D 主视口（viewport）和两个 2D 副视口（subviewport），其中次级 2D 副视口用于显示二维切片图。

场景的相机位置就是我们看图表的视角，当旋转一个图表时，实际上就是相机位置的变化，Qt 为相机提供了一些预设的视角，由枚举类型 Q3DCamera::CameraPreset 定义，它有二十多种取值，其中几种是：

- Q3DCamera::CameraPresetFrontLow，前下方；
- Q3DCamera::CameraPresetFront，正前方；
- Q3DCamera::CameraPresetFrontHigh，前上方；
- Q3DCamera::CameraPresetLeft，左侧。

在控制面板的“预设视角”的下拉列表框中列出了所有视角取值，为 currentIndexChanged() 信号编写槽函数代码如下：

```
void MainWindow::on_comboCamera_currentIndexChanged(int index)
{ //变换视角
    Q3DCamera::CameraPreset cameraPos=Q3DCamera::CameraPreset(index);
    graph3D->scene()->activeCamera()->setCameraPreset(cameraPos);
}
```

通过 graph3D->scene()->activeCamera() 获取图表视图当前的相机，是一个 Q3DCamera 类型的对象，然后调用其 setCameraPreset() 函数设置预设的视角。因为列表框中的选项是完全按照枚举类型 Q3DCamera::CameraPreset 的取值顺序排列的，所以，只需将其序号转换为 Q3DCamera::CameraPreset 类型即可。

## 2. 旋转和缩放

在程序运行时，在图表上滚动鼠标滚轮就可以缩放，按住鼠标右键上下左右移动可以对图表进行三维旋转。图表的旋转和缩放是通过改变图表场景的相机的位置和缩放系数来实现的。

控制面板上有 3 个 QSlider 组件分别用于水平旋转、垂直旋转和缩放。3 个 QSlider 组件的 valueChanged(int value) 信号的响应代码相同，代码如下：

```
void MainWindow::on_sliderH_valueChanged(int value)
{
    Q_UNUSED(value);
    int xRot =ui->sliderH->value(); //水平
    int yRot=ui->sliderV->value(); //垂直
    int zoom=ui->sliderZoom->value(); //缩放
    graph3D->scene()->activeCamera()->setCameraPosition(xRot, yRot, zoom);
}
```

Q3DCamera 类的 setCameraPosition() 函数用于设置相机的位置和缩放系数，setCameraPosition() 函数的原型定义是：

```
void Q3DCamera::setCameraPosition(float horizontal, float vertical, float zoom=100.0f)
```

其中，horizontal 是水平旋转角度，在  $-180^{\circ}$  至  $+180^{\circ}$  之间取值，所以 sliderH 的取值范围设置为  $-180$  至  $+180$ ；vertical 是垂直方向旋转角度，在  $0$  至  $90^{\circ}$  之间取值，所以 sliderV 的取值范围设置为  $0$  至  $90$ ；zoom 是缩放系数，缺省值为 100，表示无缩放，sliderZoom 的取值范围设置为 10

至 500，小于 100 是缩小，大于 100 是放大。

setCameraPosition()函数同时定义了相机的位置和缩放，所以 3 个 QSlider 组件的 valueChanged(int value)信号响应代码相同。

### 3. 主题

与 QChart 一样，QAbstract3DGraph 类可以设置主题，主题定义了图表的各种颜色和外观设置。通过 activeTheme()可以获取图表当前设置的主题，通过 setActiveTheme()函数可以设置新的主题。

用于三维图表的主题类是 Q3DTheme，其 type()属性表示主题类型。主题类型是一个枚举类型 Q3DTheme::Theme，有多种取值，与 QChart 的主题的枚举类型的取值类似。

控制面板上的“主题”下拉列表框中列出了所有可选的主题，为其 currentIndexChanged(int index)信号编写响应槽函数，代码如下：

```
void MainWindow::on_cBoxTheme_currentIndexChanged(int index)
{ // 设置主题
    Q3DTheme *currentTheme = graph3D->activeTheme();
    currentTheme->setType(Q3DTheme::Theme(index));
}
```

图表的一些显示属性的设置是通过设置 activeTheme()主题的属性来实现的，例如控制面板上的“显示背景”复选框的代码如下：

```
void MainWindow::on_chkBoxBackground_clicked(bool checked)
{ // 图表的背景
    graph3D->activeTheme()->setBackgroundEnabled(checked);
}
```

“显示背景网格”复选框的代码如下：

```
void MainWindow::on_chkBoxGrid_clicked(bool checked)
{ // 图表的网格
    graph3D->activeTheme()->setGridEnabled(checked);
}
```

“坐标轴标签背景可见”的复选框的代码如下：

```
void MainWindow::on_chkBoxAxisBackground_clicked(bool checked)
{ // 轴标签背景
    graph3D->activeTheme()->setLabelBackgroundEnabled(checked);
}
```

### 4. 选择模式

“选择模式”指的是鼠标在图表上单击时，项被选择的模式，缺省为选择一个项。QAbstract3DGraph 类的 setSelectionMode()函数用于设置选择模式，选择模式是枚举类型 QAbstract3DGraph::SelectionFlag，其取值常量见表 10-1。

表 10-1 QAbstract3DGraph::SelectionFlag 枚举类型取值

枚举常量	意义
SelectionNone	不允许选择
SelectionItem	选择并且高亮度显示一个项
SelectionRow	选择并且高亮度显示一行
SelectionItemAndRow	选择一个项和一行，用不同颜色高亮显示

续表

枚举常量	意义
SelectionColumn	选择并且高亮度显示一列
SelectedItemAndColumn	选择一个项和一列，用不同颜色高亮显示
SelectionRowAndColumn	选择交叉的一行和一列
SelectionItemRowAndColumn	选择交叉的一行和一列，用不同颜色高亮显示
SelectionSlice	切片选择，需要与 SelectionRow 或 SelectionColumn 结合使用
SelectionMultiSeries	选中同一个位置处的多个序列的项

控制面板的“选择模式”下拉列表框中列出了除 SelectionMultiSeries 之外的其他所有选择模式，其 currentIndexChanged(int index)信号的槽函数代码如下：

```
void MainWindow::on_cBoxSelectionMode_currentIndexChanged(int index)
{ //选择模式
    graph3D->setSelectionMode(QAbstract3DGraph::SelectionMode(index));
}
```

### 5. 序列相关的设置

Q3DBars 只能显示 QBar3DSeries 序列，QBar3DSeries、QScatter3DSeries 和 QSurface3DSeries 都继承自 QAbstract3DSeries（见图 10-2）。

序列的设置主要是对其一些显示属性的设置，QAbstract3DSeries 类的主要函数见表 10-2（省略了函数参数中的 const 关键字）。一个设置函数一般对应一个读取函数，如 setBaseColor()用于设置序列基本颜色，对应的读取序列基本颜色的函数是 baseColor()，表中仅列出设置函数。

表 10-2 QAbstract3DSeries 类的主要功能函数

函数原型	功能描述
void setBaseColor(QColor &color)	设置序列基本颜色
void setBaseGradient(QLinearGradient &gradient)	设置序列渐变色
void setColorStyle(Q3DTheme::ColorStyle style)	设置序列颜色类型
void setMesh(Mesh mesh)	设置棒图项的样式，参数是 QAbstract3DSeries::Mesh 枚举类型，定义棒图是棱柱、圆柱、圆锥形等
void setMeshSmooth(bool enable)	设置棒图是否有光滑效果
void setMeshRotation(QQuaternion &rotation)	设置所有项的旋转角度
void setItemLabelVisible(bool visible)	设置项的标签是否可见，若为 True，则选中一个项时会显示其标签
void setName(QString &name)	设置序列的名称，标记符号“@seriesName”可用于项的标签格式定义中
void setVisible(bool visible)	设置序列是否可见
void setItemLabelFormat(QString &format)	设置选中项的标签文字格式

窗口控制面板中的“棒图样式”下拉列表框列出了 QAbstract3DSeries::Mesh 枚举类型的各种取值，用于设置棒图的样式，如：

- QAbstract3DSeries::MeshBar，棱柱；
- QAbstract3DSeries::MeshCylinder，圆柱；
- QAbstract3DSeries::MeshSphere，椭圆。

还有一些其他取值，其详细描述见 Qt 的帮助文件。

“棒图样式”下拉列表框的 currentIndexChanged(int index)信号的槽函数代码如下：

```
void MainWindow::on_cBoxBarStyle_currentIndexChanged(int index)
```

```

{ //棒图的样式
    QAbstract3DSeries::Mesh aMesh;
    aMesh=QAbstract3DSeries::Mesh(index+1);
    series->setMesh(aMesh);
}

```

“项的标签可见”复选框的响应代码如下：

```

void MainWindow::on_chkBoxItemLabel_clicked(bool checked)
{ //项的标签是否可见
    series->setItemLabelFormat("value at (@rowLabel,@colLabel): %.1f");
    series->setItemLabelVisible(checked);
}

```

这里使用了 `setItemLabelFormat()` 函数设置项的标签显示文字的格式，在格式设置中，可以使用一些标记符号，见表 10-3。

表 10-3 QBar3DSeries 的 `setItemLabelFormat()` 函数可用的标记符号

标记符号	意义
@rowTitle	行坐标轴的标题
@colTitle	列坐标轴的标题
@valueTitle	数值坐标轴的标题
@rowIdx	可见的行索引号
@colIdx	可见的列索引号
@rowLabel	项所在的行坐标的文字标签
@colLabel	项所在的列坐标的文字标签
@valueLabel	项的数值，显示格式与 <code>QValue3DAxis::labelFormat</code> 格式相同
@seriesName	序列名称
%<format spec>	指定的数值显示格式，格式规则与 <code>QValue3DAxis::labelFormat</code> 相同

## 6. 坐标轴反向

在图 10-5 的三维柱状图中，垂直方向是数值坐标轴，缺省的方向是向上为正。如果实际情况需要数值坐标轴反向，可以调用 `QValue3DAxis::setReversed()` 函数将坐标轴设置为反向。控制面板上的“数值坐标轴反向”复选框的槽函数代码如下：

```

void MainWindow::on_chkBoxReverse_clicked(bool checked)
{ //数值轴反向
    graph3D->valueAxis()->setReversed(checked);
}

```

# 10.3 三维散点图

## 10.3.1 绘制三维散点图

绘制三维散点图采用 `Q3DScatter` 类，绘图序列是 `QScatter3DSeries` 类。实例 `samp10_2` 演示如何绘制三维散点图，实例运行界面如图 10-7 所示，绘制了一个“墨西哥草帽”的散点图。

程序的界面与实例 `samp10_1` 类似，左侧是控制面板，右侧是绘图区，项目的创建和界面元素的布局也相同。

主窗口类 `MainWindow` 的主要定义如下（省略了界面组件的槽函数定义）：



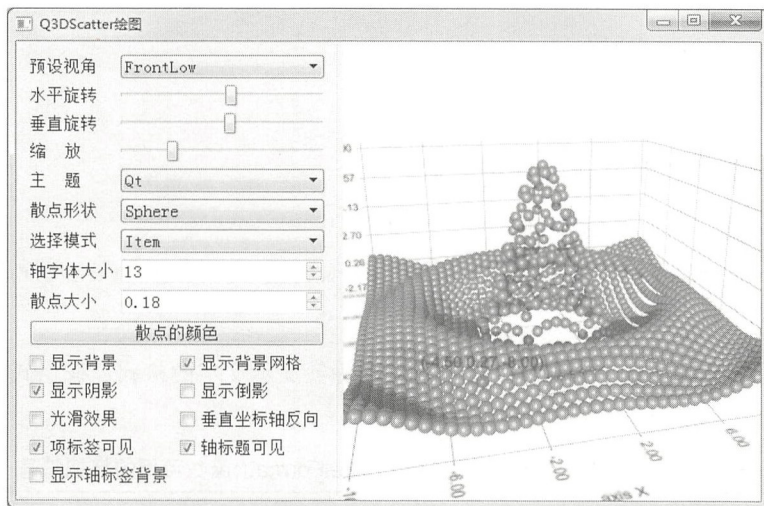


图 10-7 实例 samp10\_2 运行效果

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QWidget *graphContainer; //图表的容器
    Q3DScatter *graph3D; //散点图
    QScatter3DSeries *series; //散点序列
    void iniGraph3D(); //初始化绘图
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    Ui::MainWindow *ui;
};
```

在其中定义了绘制三维散点图的 Q3DScatter 类图表对象 graph3D，以及绘图序列 QScatter3DSeries 类的散点序列 series。

iniGraph3D()函数用于图表的创建，在构造函数里完成图表的创建。下面是主窗口构造函数和 iniGraph3D()函数的代码。

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    iniGraph3D();
    QSplitter *splitter=new QSplitter(Qt::Horizontal);
    splitter->addWidget(ui->groupBox);
    splitter->addWidget(graphContainer);
    this->setCentralWidget(splitter);
}
void MainWindow::iniGraph3D()
{
    graph3D = new Q3DScatter();
    graphContainer = QWidget::createWindowContainer(graph3D);
```

```

QScatterDataProxy *proxy = new QScatterDataProxy(); //数据代理
series = new QScatter3DSeries(proxy); //创建序列
series->setItemLabelFormat("(xLabel yLabel zLabel)");
series->setMeshSmooth(true);
graph3D->addSeries(series);
//创建坐标轴
graph3D->axisX()->setTitle("axis X");
graph3D->axisX()->setTitleVisible(true);
graph3D->axisY()->setTitle("axis Y");
graph3D->axisY()->setTitleVisible(true);
graph3D->axisZ()->setTitle("axis Z");
graph3D->axisZ()->setTitleVisible(true);
graph3D->activeTheme()->setLabelBackgroundEnabled(false);
series->setMesh(QAbstract3DSeries::MeshSphere); //数据点为圆球
series->setItemSize(0.2); //取值范围 0~1

int N=41;
int itemCount=N*N;
QScatterDataArray *dataArray = new QScatterDataArray();
dataArray->resize(itemCount);
QScatterDataItem *ptrToDataArray = &dataArray->first();
//墨西哥草帽, -10:0.5:10, N=41
float x,y,z;
int i,j;
x=-10;
for (i=1 ; i <=N; i++)
{
    y=-10;
    for ( j =1; j <=N; j++)
    {
        z=qSqrt(x*x+y*y);
        if (z!=0)
            z=10*qSin(z)/z;
        else
            z=10;
        ptrToDataArray->setPosition(QVector3D(x,z,y));
        ptrToDataArray++;
        y+=0.5;
    }
    x+=0.5;
}
series->dataProxy()->resetArray(dataArray);
}

```

在 `iniGraph3D()` 函数中, 程序显式地创建了数据代理组件, 并在序列创建时将此数据代理传递给序列。

```

QScatterDataProxy *proxy = new QScatterDataProxy(); //数据代理
series = new QScatter3DSeries(proxy); //创建序列

```

每一种序列都有其对应的数据代理类, 与散点序列对应的数据代理类是 `QScatterDataProxy`。

散点序列的每一个点都是一个 `QScatterDataItem` 类, 它存储一个散点的三维坐标和旋转角度信息。`QScatterDataArray` 是 `QScatterDataItem` 类的向量的类型定义, 定义如下:

```

QVector<QScatterDataItem> QScatterDataArray;

```

所以, `QScatterDataArray` 实际就是一个一维数组, 数组的元素是 `QScatterDataItem` 对象。创建 `QScatterDataArray` 对象 `dataArray`, 并预设其大小。

```
QScatterDataArray *dataArray = new QScatterDataArray();
dataArray->resize(itemCount);
```

程序中根据数学公式计算“墨西哥草帽”每个散点的坐标, 并且保存到数组 `dataArray` 里, 最后将此数组设置为数据代理的数据内容, 即:

```
series->dataProxy()->resetArray(dataArray);
```

**注意** 在此实例中, “墨西哥草帽”的数据点在水平面上是均匀分布的, 所以相当于在水平面上做了网格划分, 每个网格里面都有一个数据点。但是散点图并不要求数据点规则分布, 它只是根据每个散点的三维坐标和旋转方向绘图。

### 10.3.2 三维坐标轴的方向

由于三维散点图是根据数据点的三维坐标绘图, 所以 3 个坐标轴都是 `QValue3DAxis` 类型的坐标轴, 在 `iniGraph3D()` 函数中, 为 3 个坐标轴标注了轴标题, 在绘制出来的三维图形中会发现, 从正前方看所绘制的三维图时, 3 个坐标轴的正向分布如图 10-8 所示。

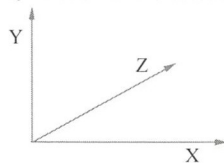


图 10-8 三维图缺省的坐标方向

在此坐标系中, 东西方向为  $X$  轴, 南北方向为  $Z$  轴, 且向北方向为正, 垂直方向为  $Y$  轴, 向上为正。所以, 在 `iniGraph3D()` 函数中, 计算墨西哥草帽的散点坐标时采用的是笛卡尔坐标, 水平面是  $X$  和  $Y$ , 垂直方向是  $Z$ , 所以计算出的  $(x, y, z)$  坐标添加作为散点的坐标时, 使用的语句如下:

```
ptrToDataArray->setPosition(QVector3D(x, z, y));
```

### 10.3.3 散点形状与大小

散点形状使用 `QAbstract3DSeries` 类的 `setMesh()` 函数进行设置, 设置的参数类型是枚举类型 `QAbstract3DSeries::Mesh`。这个函数在柱状图的设置中, 是设置每个项的形状; 在散点图中用于设置每个散点的形状。

散点的大小由 `QScatter3DSeries` 类的函数 `setItemSize(float size)` 进行设置, 参数 `size` 是 0 至 1 之间的小数, 表示散点的相对大小, 图表会根据数值自动调整散点的大小。

控制面板上其他的设置与柱状图实例 `samp10_1` 中的一样, 这里不再详述。

## 10.4 三维曲面绘图

### 10.4.1 三维曲面图

#### 1. 绘制三维曲面图

绘制三维曲面使用 `Q3DSurface` 图表类和 `QSurface3DSeries` 序列, 根据使用的数据代理类的不同

同，可以绘制两种三维曲面图。

- QSurfaceDataProxy 数据代理类根据空间点的三维坐标绘制曲面，如一般的三维函数曲面。
- QHeightMapSurfaceDataProxy 数据代理类根据一个图片的数据绘制三维曲面，典型的如三维地形图。

图 10-9 是实例 samp10\_3 使用 QSurfaceDataProxy 数据代理绘制普通三维曲面图的运行界面，左侧的控制面板的控制功能与前述实例基本相同，只是增加了 3 个颜色设置的按钮。

下面是主窗口类的定义的主要部分（省略了界面组件的槽函数定义）：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QWidget          *graphContainer;
    Q3DSurface       *graph3D;    //三维图表
    QSurface3DSeries *series;     //序列
    QSurfaceDataProxy *proxy;     //数据代理
    void iniGraph3D();
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    Ui::MainWindow *ui;
};
```

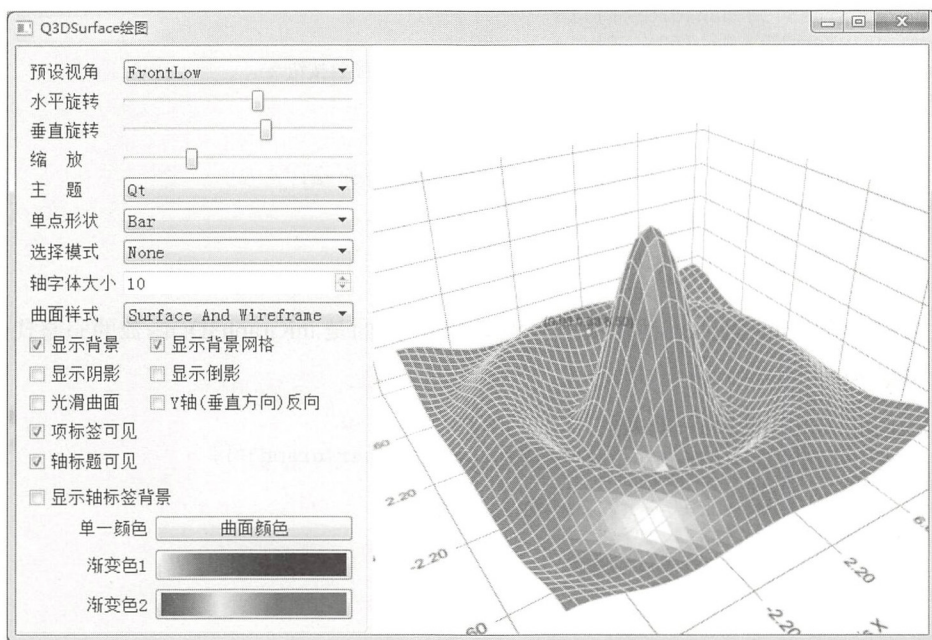


图 10-9 三维曲面图

MainWindow 类定义了 Q3DSurface 类的变量 graph3D，QSurface3DSeries 类的序列变量 series，QSurfaceDataProxy 类的的数据代理变量 proxy。iniGraph3D()函数用于图表初始化绘制，在构造函数



里调用此函数。

下面是 MainWindow 类的构造函数，在构造函数里为控制面板上的两个设置曲面渐变颜色的按钮绘制了渐变图片。QLinearGradient 类的使用见 8.1 节，这里不再赘述。

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    //渐变颜色按钮
    QLinearGradient grBtoY(0, 0, 100, 0); //线性渐变
    grBtoY.setColorAt(1.0, Qt::black);
    grBtoY.setColorAt(0.67, Qt::blue);
    grBtoY.setColorAt(0.33, Qt::red);
    grBtoY.setColorAt(0.0, Qt::yellow);
    QPixmap pm(160, 20); //图片
    QPainter pmp(&pm);
    pmp.setBrush(QBrush(grBtoY)); //渐变颜色 1
    pmp.setPen(Qt::NoPen);
    pmp.drawRect(0, 0, 160, 20);
    ui->btnGrad1->setIcon(QIcon(pm)); //渐变颜色按钮 1
    ui->btnGrad1->setIconSize(QSize(160, 20));

    QLinearGradient grGtoR(0, 0, 100, 0); //渐变颜色 2
    grGtoR.setColorAt(1.0, Qt::darkGreen);
    grGtoR.setColorAt(0.5, Qt::yellow);
    grGtoR.setColorAt(0.2, Qt::red);
    grGtoR.setColorAt(0.0, Qt::darkRed);
    pmp.setBrush(QBrush(grGtoR));
    pmp.drawRect(0, 0, 160, 20);
    ui->btnGrad2->setIcon(QIcon(pm)); //渐变颜色按钮 2
    ui->btnGrad2->setIconSize(QSize(160, 20));

    iniGraph3D();
    QSplitter *splitter=new QSplitter(Qt::Horizontal);
    splitter->addWidget(ui->groupBox);
    splitter->addWidget(graphContainer);
    this->setCentralWidget(splitter);
}
```

构造函数里调用 iniGraph3D()函数进行图形绘制，下面是 iniGraph3D()函数的完整代码。

```
void MainWindow::iniGraph3D()
{
    graph3D = new Q3DSurface();
    graphContainer = QWidget::createWindowContainer(graph3D);
    //创建坐标轴
    QValue3DAxis *axisX=new QValue3DAxis;
    axisX->setTitle("Axis X");
    axisX->setTitleVisible(true);
    axisX->setRange(-11,11);
    graph3D->setAxisX(axisX);

    QValue3DAxis *axisY=new QValue3DAxis;
    axisY->setTitle("Axis Y");
    axisY->setTitleVisible(true);
    axisY->setAutoAdjustRange(true);
```

```

graph3D->setAxisY(axisY);
QValue3DAxis *axisZ=new QValue3DAxis;
axisZ->setTitle("Axis Z");
axisZ->setTitleVisible(true);
axisZ->setRange(-11,11);
graph3D->setAxisZ(axisZ);
//创建数据代理
proxy = new QSurfaceDataProxy();
series = new QSurface3DSeries(proxy);
graph3D->addSeries(series);
series->setItemLabelFormat("(xLabel yLabel zLabel)");
series->setMeshSmooth(true);
graph3D->activeTheme()->setLabelBackgroundEnabled(false);
series->setDrawMode(QSurface3DSeries::DrawSurfaceAndWireframe);
//创建数据, 墨西哥草帽
int N=41;//-10:0.5:10, N 个数据点
QSurfaceDataArray *dataArray = new QSurfaceDataArray; //数组
dataArray->reserve(N);
float x=-10,y,z;
int i,j;
for (i =1 ; i <=N; i++)
{
    QSurfaceDataRow *newRow = new QSurfaceDataRow(N); //一行的数据
    y=-10;
    int index=0;
    for (j = 1; j <=N; j++)
    {
        z=qSqrt(x*x+y*y);
        if (z!=0)
            z=10*qSin(z)/z;
        else
            z=10;
        (*newRow)[index++].setPosition(QVector3D(x, z, y));
        y=y+0.5;
    }
    x=x+0.5;
    *dataArray << newRow;
}
proxy->resetArray(dataArray);
}

```

代码首先创建三维图表 `graph3D`, 并为其创建窗口容器, 然后创建了 3 个 `QValue3DAxis` 类型的坐标轴对象, 设置基本属性后作为图表的相应坐标轴。

程序创建 `QSurfaceDataProxy` 类型的数据代理变量 `proxy`, 并在创建序列时作为参数传递给它, 然后将序列添加到图表。

下面的代码对序列做了一些属性的设置, 特别设置了绘图模式为曲面加线网的模式, 即:

```
series->setDrawMode(QSurface3DSeries::DrawSurfaceAndWireframe);
```

然后是创建数据, 采用与实例 `samp10_2` 相同的“墨西哥草帽”函数生成数据点。`QsurfaceDataProxy` 数据代理采用两层一维数组的模式存储三维曲面的数据, 与三维柱状图的数据代理类存储数据的模式类似。

这里创建一个 `QSurfaceDataArray` 类型的向量 `dataArray`, 它的元素个数等于行数, 每一行又

是一个 `QSurfaceDataRow` 类型的列表，用于存储一行中多列的数据。`QSurfaceDataArray` 和 `QSurfaceDataRow` 是类型定义，其定义为：

```
typedef QVector<QSurfaceDataItem> QSurfaceDataRow;
typedef QList<QSurfaceDataRow *> QSurfaceDataArray;
```

每个基本的数据点都是 `QSurfaceDataItem` 类，有  $x$ 、 $y$ 、 $z$  3 个坐标，通过 `setPosition(const QVector3D &pos)` 函数可以设置一个点的三维坐标。同样，`Q3DSurface` 的三维坐标系是图 10-8 的样式，与在数学中常见的坐标轴的名称不一致，在设置数据点时需要注意。

## 2. 曲面样式

`QSurface3DSeries` 的函数 `setDrawMode(DrawFlags mode)` 用于设置曲面样式，参数是枚举类型 `QSurface3DSeries::DrawFlag` 的组合，取值有以下几种：

- `QSurface3DSeries::DrawWireframe`，只绘制线网；
- `QSurface3DSeries::DrawSurface`，只绘制曲面；
- `QSurface3DSeries::DrawSurfaceAndWireframe`，绘制线网和曲面。

图 10-9 是设置为 `QSurface3DSeries::DrawSurfaceAndWireframe` 时的绘图效果。

## 3. 曲面的单一颜色设置

在改变图表的主题时，曲面的颜色会根据主题的设置而改变，也可以单独改变曲面的颜色。控制面板中的“曲面颜色”按钮用于设置曲面颜色，代码如下：

```
void MainWindow::on_btnBarColor_clicked()
{ //单一曲面颜色
    QColor color=series->baseColor();
    color=QColorDialog::getColor(color);
    if (color.isValid())
    {
        series->setBaseColor(color);
        series->setColorStyle(Q3DTheme::ColorStyleUniform);
    }
}
```

对序列调用 `setBaseColor()` 函数设置颜色，并且调用 `setColorStyle()` 函数将序列颜色样式设置为单一颜色。`setColorStyle()` 设置的参数是枚举类型 `Q3DTheme::ColorStyle`，其取值有以下几种：

- `Q3DTheme::ColorStyleUniform`，单一颜色；
- `Q3DTheme::ColorStyleObjectGradient`，不考虑对象高度，只根据对象渐变；
- `Q3DTheme::ColorStyleRangeGradient`，根据对象高度渐变。

## 4. 曲面的渐变颜色设置

曲面还可以设置为渐变颜色，例如 `Q3DTheme::ColorStyleRangeGradient` 渐变类型，即根据数据点的高度赋予不同的颜色。控制面板中“渐变色 1”按钮的响应代码如下：

```
void MainWindow::on_btnGrad1_clicked()
{ //渐变颜色
    QLinearGradient gr;
    gr.setColorAt(0.0, Qt::black);
    gr.setColorAt(0.33, Qt::blue);
    gr.setColorAt(0.67, Qt::red);
```

```

gr.setColorAt(1.0, Qt::yellow);
series->setBaseGradient(gr);
series->setColorStyle(Q3DTheme::ColorStyleRangeGradient);
}

```

程序定义一个 `QLinearGradient` 线性渐变类的变量，定义了渐变颜色属性后，调用序列的 `setBaseGradient()` 函数设置渐变颜色，并设置颜色样式为 `Q3DTheme::ColorStyleRangeGradient`。

## 10.4.2 三维地形图

### 1. 绘制三维地形图

当为 `QSurface3DSeries` 序列使用 `QHeightMapSurfaceDataProxy` 数据代理时，数据代理可以从一个含有高程数据的图片中读入数据，绘制地形图。图 10-10 是实例 `samp10_4` 运行界面，右侧的三维曲面是一个地形图。

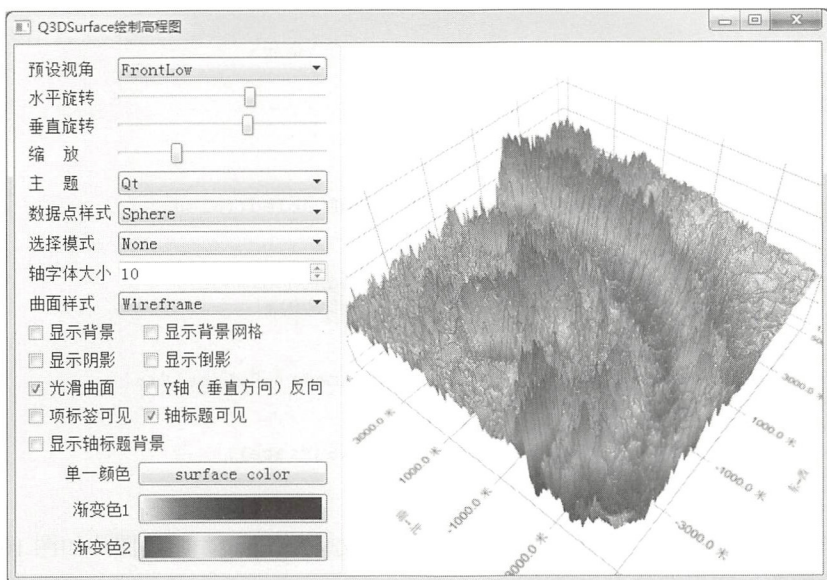


图 10-10 实例 `samp10_4` 运行界面

实例 `samp10_4` 的程序结构与实例 `samp10_3` 相同，只是数据代理类定义为 `QHeightMapSurfaceDataProxy` 类，下面是 `MainWindow` 类的定义部分。

```

class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QWidget      *graphContainer;
    Q3DSurface   *graph3D;          //三维图表
    QSurface3DSeries *series;       //序列
    QHeightMapSurfaceDataProxy *proxy; //数据代理
    void iniGraph3D();
public:
    explicit MainWindow(QWidget *parent = 0);
};

```



iniGraph3D()函数实现绘图，其完整代码如下：

```
void MainWindow::iniGraph3D()
{
    graph3D = new Q3DSurface();
    graphContainer = QWidget::createWindowContainer(graph3D);
    //创建坐标轴
    QValue3DAxis *axisX=new QValue3DAxis;    //X, 东西
    axisX->setTitle("东--西");
    axisX->setTitleVisible(true);
    axisX->setLabelFormat("%.1f 米");
    axisX->setRange(-5000,5000);
    graph3D->setAxisX(axisX);

    QValue3DAxis *axisY=new QValue3DAxis; //Y, 深度
    axisY->setTitle("深度");
    axisY->setTitleVisible(true);
    axisY->setAutoAdjustRange(true);
    graph3D->setAxisY(axisY);

    QValue3DAxis *axisZ=new QValue3DAxis;    //Z, 南北
    axisZ->setTitle("南--北");
    axisZ->setLabelFormat("%.1f 米");
    axisZ->setTitleVisible(true);
    axisZ->setRange(-5000,5000);
    graph3D->setAxisZ(axisZ);
    graph3D->activeTheme()->setLabelBackgroundEnabled(false);
    //创建数据代理
    QImage heightMapImage(":/map/sea.png");
    proxy = new QHeightMapSurfaceDataProxy(heightMapImage);
    proxy->setValueRanges(-5000, 5000, -5000, 5000);

    series = new QSurface3DSeries(proxy);
    series->setItemLabelFormat("(@xLabel, @zLabel): @yLabel");
    series->setFlatShadingEnabled(false);
    series->setMeshSmooth(true);
    series->setDrawMode(QSurface3DSeries::DrawSurface);
    graph3D->addSeries(series);
}
```

在项目中创建资源文件，并且导入一个高程图片文件 sea.png，原始图片如图 10-11 所示，图片的各个点的颜色与高程有关。



图 10-11 原始的高程图片

该图片用于创建 QHeightMapSurfaceDataProxy 数据代理，并且导入图片的数据。

```
QImage heightMapImage(":/map/sea.png");
proxy = new QHeightMapSurfaceDataProxy(heightMapImage);
proxy->setValueRanges(-5000, 5000, -5000, 5000);
```

QHeightMapSurfaceDataProxy 支持多种图片文件格式，可以是彩色图也可以是灰度图。由于图片信息不包含平面坐标范围，所以需要使用时使用 setValueRanges() 设置图片数据的平面坐标范围，其函数原型如下：

```
void setValueRanges(float minX, float maxX, float minZ, float maxZ)
```

高程数据从图片中读取，如果是灰度图，就将像素的颜色值作为高度，如果是彩色图，就以红、绿、蓝 3 种颜色的平均值作为高度值。使用灰度图处理速度更快，所以，最好使用 QImage::Format\_RGB32 格式的 32 位灰度图片。

## 2. 切片选择模式

使用 QAbstract3DGraph 类的 setSelectionMode(SelectionFlags mode) 设置选择模式时，可以设置为切片选择模式。控制面板上的“选择模式”组合选择框提供了 None、Item、Row Slice 和 Column Slice 4 种选择模式，组合选择框的响应代码如下：

```
void MainWindow::on_cBoxSelectionMode_currentIndexChanged(int index)
{ // 选择模式
    switch(index)
    { case 0: // none
        graph3D->setSelectionMode(QAbstract3DGraph::SelectionNone);
        break;
      case 1: // Item
        graph3D->setSelectionMode(QAbstract3DGraph::SelectionItem);
        break;
      case 2: // Row Slice
        graph3D->setSelectionMode(QAbstract3DGraph::SelectionItemAndRow
                                   | QAbstract3DGraph::SelectionSlice);
        break;
      case 3: // Column Slice
        graph3D->setSelectionMode(QAbstract3DGraph::SelectionItemAndColumn
                                   | QAbstract3DGraph::SelectionSlice);
        break;
    }
}
```

当设置为 Row Slice 或 Column Slice 选择模式时，在三维图上单击一个位置后，图表会自动切换到切片显示模式，如图 10-12 所示。

在此状态下，中间显示选择的项所在行或列的切片图，左上角显示缩小的三维图。在左上角的三维图上再单击一下，就会恢复原始的显示状态。

在 Q3DSurface 图中，可以显示多个曲面或地形图，还可以通过 addCustomItem(QCustom3DItem \*item) 函数添加自定义三维对象，QCustom3DItem 类对象可以是由其他三维建模软件建立的三维对象，由此可以显示较复杂的三维场景。

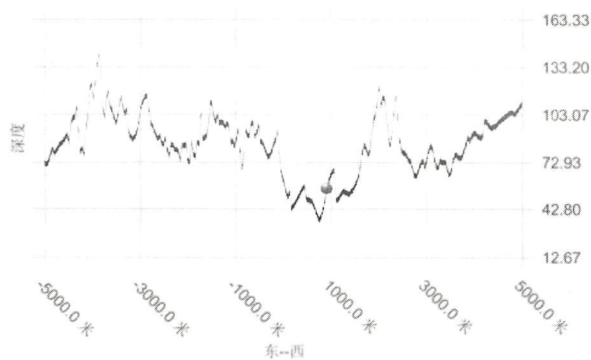


图 10-12 切片显示状态

Qt SQL 模块提供数据库编程的支持，Qt 支持多种常见的数据库，如 MySQL、Oracle、MS SQL Server、SQLite 等。Qt SQL 模块包括多个类，可以实现数据库连接、SQL 语句执行、数据获取与界面显示等功能，数据与界面之间使用 Model/View 架构，从而可以方便地实现数据的界面显示和操作。

## 11.1 Qt SQL 模块概述

要在项目中使用 Qt SQL 模块，需在项目配置文件中增加下面一条设置语句：

```
Qt += sql
```

在头文件或源程序文件中使用 Qt SQL 模块中的类，可以使用包含语句：

```
#include <QtSql>
```

这样会将 Qt SQL 模块中的所有类都包含进去，如果只使用其中的某些类，为避免冗余可以单独包含某个类。

### 11.1.1 Qt SQL 支持的数据库

Qt SQL 提供了一些常见数据库的驱动，包括网络型数据库，如 Oracle、MS SQL Server 等，也包括简单的单机型数据库，如 SQLite。Qt SQL 提供的数据库驱动见表 11-1。

表 11-1 Qt SQL 支持的数据库

驱动名	数据库
QDB2	IBM DB2（7.1 及以上版本）数据库
QIBASE	Borland InterBase 数据库
QMYSQL	MySQL 数据库
QOCI	Oracle 调用接口驱动（Oracle Call Interface Driver）
QODBC	Open Database Connectivity(ODBC)，Microsoft 的 SQL Server 数据库，以及其他支持 ODBC 接口的数据库，如 Access
QPSQL	PostgreSQL（7.3 及以上版本）数据库
QSQLITE2	SQLite 2 数据库
QSQLITE	SQLite 3 数据库
QTDS	Sybase Adaptive Server，注意：从 Qt 4.7 开始已过时



### 11.1.2 SQLite 数据库

本书假设读者对数据库的基本概念和 SQL 语句已经有基础的知识，为了用实例研究 Qt SQL 的数据库编程功能，本章采用 SQLite 数据库作为数据库实例。

SQLite 是一种无需服务器、无需进行任何配置的数据库，所有的数据表、索引等数据库元素全都存储在一个文件里，在应用程序里使用 SQLite 数据库就完全可以当作一个文件来使用。SQLite 是可以跨平台使用的数据库，在不同平台之间可以随意复制数据库。SQLite 的驱动库文件很小，包含完整功能的驱动可以小到只有 500 KB。

SQLite 是一种开源免费使用的数据库，可以从其官网下载最新版本的数据库驱动安装文件。

SQLite Expert 是 SQLite 数据库可视化管理工具，可以从其官网下载最新的安装文件，SQLite Expert 安装文件带有 SQLite 数据库驱动，所以安装 SQLite Expert 后无需再下载安装 SQLite 数据库驱动。

在 SQLite Expert 软件里建立一个数据库 demodb.db3，在此数据库里建立 4 个表，本章的 Qt SQL 编程实例都采用这个数据库文件作为数据库实例。

SQLite Expert 软件进行数据库字段设计的界面如图 11-1 所示。

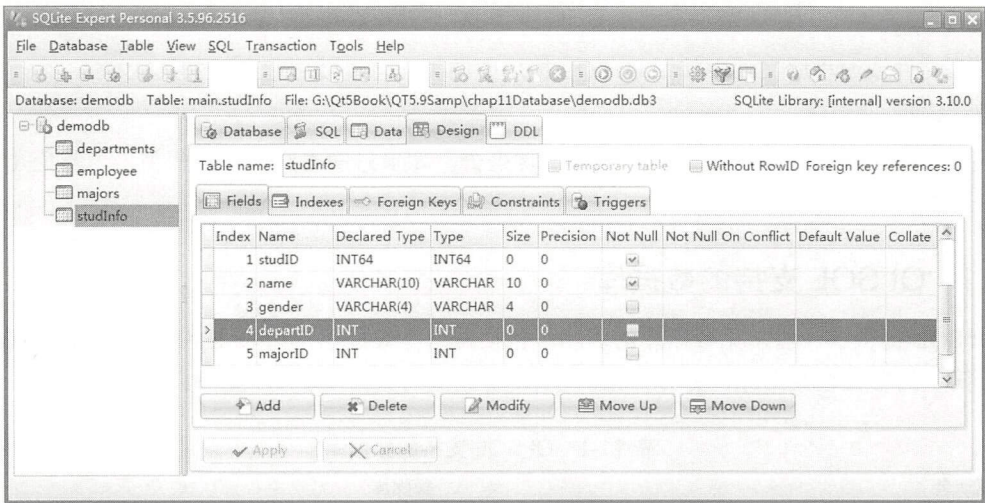


图 11-1 在 SQLite 软件里设计数据库

(1) employee 数据表是一个员工信息表，用于在实例 samp11\_1 里演示通过 QSqlTableModel 获取、显示和编辑数据表的内容，employee 的字段定义见表 11-2。

表 11-2 employee 数据表的字段定义

序号	字段名	类型	描述	说明
1	EmpNo	INT	员工编号	主键，非空
2	Name	VARCHAR(20)	姓名	非空
3	Gender	VARCHAR(4)	性别	缺省值=“男”

续表

序号	字段名	类型	描述	说明
4	Height	FLOAT	身高	缺省值=1.71
5	Birthday	DATE	出生日期	
6	Mobile	VARCHAR(18)	手机号	
7	Province	VARCHAR(20)	省份	缺省值=3500
8	City	VARCHAR(20)	城市	
9	Department	VARCHAR(30)	工作部门	
10	Education	VARCHAR(16)	教育程度	BLOB 字段可存储任何二进制内容
11	Salary	CURRENCY	工资	
12	Photo	BLOB	照片	
13	Memo	MEMO	备注	MEMO 字段可存储任意长度普通文本

(2) departments 数据表是一个学院信息表，记录学院编号和学院名称。其字段定义见表 11-3。

表 11-3 departments 数据表的字段定义

序号	字段名	类型	描述	说明
1	departID	INT	学院编号	主键，非空
2	departments	VARCHAR(40)	学院名称	非空

(3) majors 数据表是专业信息表，记录各专业的信息。其字段定义见表 11-4。

表 11-4 majors 数据表的字段定义

序号	字段名	类型	描述	说明
1	majorID	INT	专业编号	主键，非空
2	major	VARCHAR(40)	专业名称	非空
3	departID	INT	所属学院编号	非空，等于 departments 表中某个学院的 departID

departments 和 majors 构成一个 Master/Detail 关系数据表，majors 表里的 departID 字段记录了这个专业属于哪个学院，departments 表里的一条记录关联 majors 表中的多条记录。

(4) studInfo 是一个记录学生信息的数据表。其字段定义见表 11-5。

表 11-5 studInfo 数据表的字段定义

序号	字段名	类型	描述	说明
1	studID	INT	学号	主键，非空
2	name	VARCHAR(10)	姓名	非空
3	gender	VARCHAR(4)	性别	
4	departID	INT	学院编号	关联 departments 表中的记录
5	majorID	INT	专业编号	关联 majors 表中的记录

studInfo 表中记录学生的所在学院采用了代码字段 departID，具体的学院名称需要通过查询 departments 表中相同的 departID 的记录获得；majorID 记录了专业代码，具体的专业名称需要查找 majors 表中的记录获取。这两个字段都是代码字段，只存储代码，具体的意义需要查询关联数据表的相应记录获得，在实际的数据库设计中经常用到这种设计方式。

Qt SQL 中使用 QSqlRelationalTableModel 可以很方便地实现这种代码型数据表的显示与编辑，实例 samp11\_4 显示了 QSqlRelationalTableModel 的使用方法。

### 11.1.3 Qt SQL 模块的主要类

Qt SQL 提供的主要类的简要功能描述见表 11-6。

表 11-6 Qt SQL 模块包含的主要类的功能

类名称	功能描述
QSqlDatabase	用于建立与数据库的连接
QSqlDriver	用于访问具体的 SQL 数据库的底层抽象类
QSqlDriverCreator	为某个具体的数据库驱动提供 SQL 驱动的模板类
QSqlDriverCreatorBase	所有 SQL 驱动器的基类
QSqlDriverPlugin	用于定制 QSqlDriver 插件的抽象基类
QSqlError	SQL 数据库错误信息，可以用于访问上一次出错的信息
QSqlField	操作数据表或视图（view）的字段类
QSqlIndex	操作数据库的索引类
QSqlQuery	执行各种 SQL 语句的类
QSqlQueryModel	SQL 查询结果数据的只读数据模型（data model），用于 SELECT 查询结果数据记录的只读显示
QSqlRecord	封装了数据记录操作的类
QSqlRelation	用于存储 SQL 外键信息的类，用于 QSqlRelationalTableModel 数据源中设置代码字段与关联数据表的关系
QSqlRelationalDelegate	用于 QSqlRelationalTableModel 的一个代码字段的显示和编辑代理组件，一般是一个 QComboBox 组件，下拉列表中自动填充代码表的代码字段对应的实际内容
QSqlRelationalTableModel	用于一个数据表的可编辑的数据模型，支持代码字段的外键
QSqlResult	访问 SQL 数据库的抽象接口
QSqlTableModel	编辑一个单一数据表的数据模型类
QDataWidgetMapper	用于界面组件与字段之间实现映射，实现字段内容自动显示的类

QSqlDatabase 用于建立与数据库的连接，一般是先加载需要的数据库驱动，然后设置数据库的登录参数，如主机地址、用户名、登录密码等，如果是单机型数据库，如 SQLite，只需设置数据库文件即可。

数据库的操作一般需要将数据库的内容在界面上进行显示和编辑，Qt 采用 Model/View 结构进行数据库内容的界面显示。QTableView 是常用的数据库内容显示视图组件，用于数据库操作的数据模型类有 QSqlQueryModel、QSqlTableModel 和 QSqlRelationalTableModel，这几个类的继承关系见图 11-2。

QSqlQueryModel 通过设置 SELECT 语句查询获取数据库的内容，但是 QSqlQueryModel 的数据是只读的，不能进行编辑。

QSqlTableModel 直接设置一个数据表的名称，可以获取数据表的全部记录，其结果是可编辑的，设置为界面上的 QTableView 组件的数据模型后就可以显示和编辑数据。

QSqlRelationalTableModel 编辑一个数据表，并且可以将代码字段通过关系与代码表关联，将代码字段的编辑转换为直观的内容选择编辑。

QSqlQuery 是另外一个经常使用的类，它可以执行任何 SQL 语句，特别是没有返回记录的语句，如 UPDATE、INSERT、DELETE 等，通过 SQL 语句对数据库直接进行编辑修改。

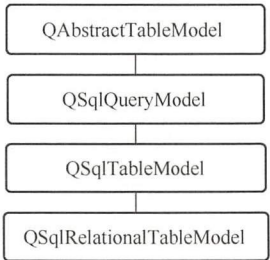


图 11-2 数据库相关数据模型类的继承关系



## 11.2 QSqlTableModel 的使用

### 11.2.1 实例功能

实例 samp11\_1 使用 QSqlTableModel 显示实例数据库 demodb 中 employee 数据表的内容, 实现编辑、插入、删除记录的操作, 实现数据的排序和记录过滤, 还实现 BLOB 类型字段 Photo 中存储照片的显示、导入等操作, 运行界面如图 11-3 所示。

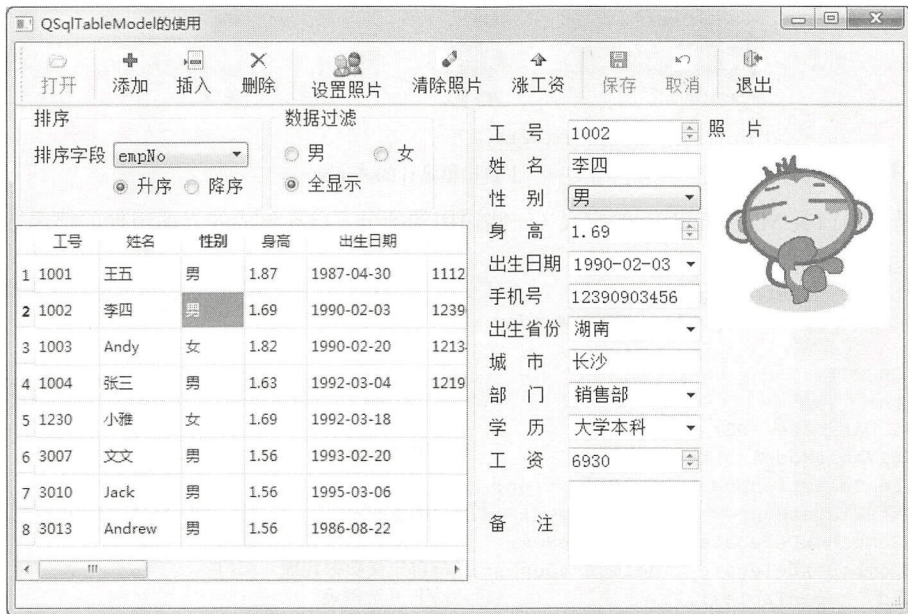


图 11-3 实例 samp11\_1 运行界面

在图 11-3 的中, 左侧数据表格是一个 QTableView 组件, 设置一个 QSqlTableModel 类的变量作为数据模型后就可以显示数据表的内容。右侧的一些编辑框、下拉列表框等。界面组件通过 QDataWidgetMapper 类的实例设置为与某个字段关联, 自动显示字段的内容。但是没有现成的组件可以通过字段映射显示 Photo 字段中的照片。为此使用一个 QLabel 组件, 通过其 pixmap 属性显示图片, 所以照片的显示、导入和清除需要单独操作, 并且与数据模块的记录移动保持同步, 如同将此界面组件与 Photo 字段映射了一样。

QTableView 显示内容有缺省的代理组件, 一般是自动使用 QLineEdit 组件。但是对于某些字段期望通过下拉列表框来选择输入, 例如“性别”和“部门”这两个字段。为此, 还设计了自定义数据代理类, “性别”和“部门”两个字段使用 QComboBox 进行编辑。

工具栏上的按钮根据当前状态自动可用或禁用, 特别是“保存”和“取消”两个按钮在数据表的内容被修改后自动变为可用, 当保存或取消修改后又变为不可用。



## 11.2.2 主窗口设计

实例 samp11\_1 是一个主窗口继承自 QMainWindow 的应用程序,使用 UI 设计器可视化设计界面。设计的 Action 如图 11-4 所示,使用 Action 创建工具栏按钮,其他界面组件和布局不再详述。

Name	Used	Text	Shortcut	Checkable	ToolTip
 actOpenDB	<input checked="" type="checkbox"/>	打开		<input type="checkbox"/>	打开数据库
 actQuit	<input checked="" type="checkbox"/>	退出		<input type="checkbox"/>	退出
 actRecAppend	<input checked="" type="checkbox"/>	添加		<input type="checkbox"/>	添加记录
 actRecInsert	<input checked="" type="checkbox"/>	插入		<input type="checkbox"/>	插入记录
 actSubmit	<input checked="" type="checkbox"/>	保存		<input type="checkbox"/>	保存修改
 actRevert	<input checked="" type="checkbox"/>	取消		<input type="checkbox"/>	取消修改
 actRecDelete	<input checked="" type="checkbox"/>	删除		<input type="checkbox"/>	删除记录
 actPhoto	<input checked="" type="checkbox"/>	设置照片		<input type="checkbox"/>	设置照片
 actPhotoClear	<input checked="" type="checkbox"/>	清除照片		<input type="checkbox"/>	清除照片
 actScan	<input checked="" type="checkbox"/>	张工资		<input type="checkbox"/>	张工资

Action Editor    Signals & Slots Editor

图 11-4 主窗口里设计的 Action

以下是主窗口类 MainWindow 的定义(一些通用的部分,自动生成的界面组件的槽函数省略了)。

```
#include <QtSql>
#include <QDataWidgetMapper>
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QSqlDatabase DB;                //数据库连接
    QSqlTableModel *tabModel;       //数据模型
    QItemSelectionModel *theSelection; //选择模型
    QDataWidgetMapper *dataMapper;  //数据映射
    QWComboBoxDelegate delegateSex; //自定义数据代理,性别
    QWComboBoxDelegate delegateDepart; //自定义数据代理,部门
    void openTable();               //打开数据表
    void getFieldNames();           //获取字段名称,填充“排序字段”的 comboBox
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    void on_currentChanged(const QModelIndex &current, const QModelIndex &previous);
    void on_currentRowChanged(const QModelIndex &current, const QModelIndex &previous);
}
```

MainWindow 类中定义了几个私有变量。

- QSqlDatabase DB, 用于加载数据库驱动和建立与数据库之间的连接。
- QSqlTableModel \*tabModel, 用于指定某一个数据表, 作为数据表的数据模型。
- QItemSelectionModel \*theSelection, 作为 tabModel 的选择模型, 提供 currentChanged()、currentRowChanged()等信号, 在 tabModel 选择的字段发生变化、当前记录发生变化时发射信号, 以便程序进行响应。例如在 currentChanged()信号发射时, 检查 tabModel 是否有数据被修改, 从而更新界面上“保存”和“取消”两个按钮的使能状态。
- QDataWidgetMapper \*dataMapper 用于实现界面组件与 tabModel 的字段之间的映射。例如

界面上的 QLineEdit 类型的 dbEditName 组件与数据表的 Name 字段映射, 当前记录变化时会自动更新显示当前记录的 Name 字段的数据。

- QWComboBoxDelegate 是自定义的基于 QComboBox 的代理类, delegateSex 和 delegateDepart 用作 tableView 中“性别”和“部门”字段的代理组件。

私有函数 openTable() 用于打开数据表, getFieldNames() 用于获取数据表 employee 的所有字段的名称, 并填充界面上“排序字段”后的 ComboBox 组件。

定义了两个槽函数, 功能如下。

- on\_currentChanged() 用于检查数据表内容是否有修改, 从而更新“保存”和“取消”两个按钮的使能状态。
- on\_currentRowChanged() 用于在当前记录发生变化时, 从新的当前记录里提取 Photo 字段的内容, 并作为界面上的 QLabel 类型的 dblabPhoto 组件的 pixmap 显示出来。

MainWindow 的构造函数代码如下, 主要是对 tableView 的一些显示属性的设置。

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    this->setCentralWidget(ui->splitter);
    // tableView 显示属性设置
    ui->tableView->setSelectionBehavior(QAbstractItemView::SelectItems);
    ui->tableView->setSelectionMode(QAbstractItemView::SingleSelection);
    ui->tableView->setAlternatingRowColors(true);
}
```

### 11.2.3 打开数据表

工具栏上的“打开”按钮由 actOpenDB 生成, 单击此按钮时将添加 SQLite 数据库驱动、打开数据库文件、连接 employee 数据表并设置显示属性, 并且创建 tableView 显示的代理组件, 设置数据源与界面组件的映射等。其响应代码如下:

```
void MainWindow::on_actOpenDB_triggered()
{
    // 打开数据库
    QString aFile = QFileDialog::getOpenFileName(this, "选择数据库文件", "",
        "SQLite 数据库 (*.db *.db3)");
    if (aFile.isEmpty())
        return;
    DB = QSqlDatabase::addDatabase("QSQLITE"); // 添加 SQLITE 数据库驱动
    DB.setDatabaseName(aFile); // 设置数据库名称
    if (!DB.open()) // 打开数据库
    {
        QMessageBox::warning(this, "错误", "打开数据库失败",
            QMessageBox::Ok, QMessageBox::NoButton);
        return;
    }
    openTable(); // 打开数据表
}
```

由打开文件对话框选择需要打开的 SQLite 数据库文件, 本实例中需要打开 demodb.db3 数据库, 此文件在第 11 章实例程序总目录下。

私有变量 DB 是 QSqlDatabase 类。QSqlDatabase 用于数据库的操作,包括建立数据库连接,设置登录数据库的参数,打开数据库等。下面的一行语句使用 QSqlDatabase 的静态函数 addDatabase()添加 SQLite 数据库的驱动。

```
DB=QSqlDatabase::addDatabase("QSQLITE");
```

然后需要使用 QSqlDatabase 的几个函数设置数据库登录参数, setDatabaseName()设置数据库名称,对于 SQLite 数据库,就设置为数据库文件。如果是网络型数据库,如 Oracle、MS SQL Server 等,还需要使用 setHostName()设置数据库主机名, setUsername()设置数据库用户名, setPassword()设置数据库登录密码。对于 SQLite 数据库,只需用 setDatabaseName()设置数据库文件即可。

数据库连接与登录参数设置后,调用 QSqlDatabase::open()函数打开数据库。如果成功打开数据库,再调用自定义函数 openTable()打开数据表 employee,并进行相应的操作。openTable()函数代码如下:

```
void MainWindow::openTable()
{
    //打开数据表
    tabModel=new QSqlTableModel(this,DB);//数据表
    tabModel->setTable("employee");//设置数据表
    tabModel->setSort(tabModel->fieldIndex("EmpNo"),Qt::AscendingOrder);
    tabModel->setEditStrategy(QSqlTableModel::OnManualSubmit);
    if (!(tabModel->select()))//查询数据
    {
        QMessageBox::critical(this, "错误", "打开数据表错误,错误信息\n"
            +tabModel->lastError().text(),
            QMessageBox::Ok,QMessageBox::NoButton);
        return;
    }
    //设置字段显示名
    tabModel->setHeaderData(tabModel->fieldIndex("EmpNo"), Qt::Horizontal, "工号");
    tabModel->setHeaderData(tabModel->fieldIndex("Name"), Qt::Horizontal, "姓名");
    tabModel->setHeaderData(tabModel->fieldIndex("Gender"), Qt::Horizontal, "性别");
    tabModel->setHeaderData(tabModel->fieldIndex("Height"), Qt::Horizontal, "身高");
    tabModel->setHeaderData(tabModel->fieldIndex("Birthday"), Qt::Horizontal, "出生日期");
    tabModel->setHeaderData(tabModel->fieldIndex("Mobile"), Qt::Horizontal, "手机");
    tabModel->setHeaderData(tabModel->fieldIndex("Province"), Qt::Horizontal, "省份");
    tabModel->setHeaderData(tabModel->fieldIndex("City"), Qt::Horizontal, "城市");
    tabModel->setHeaderData(tabModel->fieldIndex("Department"), Qt::Horizontal, "部门");
    tabModel->setHeaderData(tabModel->fieldIndex("Education"), Qt::Horizontal, "学历");
    tabModel->setHeaderData(tabModel->fieldIndex("Salary"), Qt::Horizontal, "工资");
    tabModel->setHeaderData(tabModel->fieldIndex("Memo"), Qt::Horizontal, "备注");
    tabModel->setHeaderData(tabModel->fieldIndex("Photo"), Qt::Horizontal, "照片");
    theSelection=new QItemSelectionModel(tabModel);//选择模型
    connect(theSelection,SIGNAL(currentChanged(QModelIndex,QModelIndex)),
        this,SLOT(on_currentChanged(QModelIndex,QModelIndex)));
    connect(theSelection,SIGNAL(currentRowChanged(QModelIndex,QModelIndex)),
        this,SLOT(on_currentRowChanged(QModelIndex,QModelIndex)));

    ui->tableView->setModel(tabModel);//设置数据模型
    ui->tableView->setSelectionModel(theSelection);//设置选择模型
    ui->tableView->setColumnHidden(tabModel->fieldIndex("Memo"),true);
    ui->tableView->setColumnHidden(tabModel->fieldIndex("Photo"),true);
}
```



```

//tableView 上为“性别”和“部门”两个字段设置自定义代理组件
QStringList strList;
strList<<"男"<<"女";
bool isEditable=false;
delegateSex.setItems(strList,isEditable);
ui->tableView->setItemDelegateForColumn(
    tabModel->fieldIndex("Gender"), &delegateSex);

strList.clear();
strList<<"销售部"<<"技术部"<<"生产部"<<"行政部";
isEditable=true;
delegateDepart.setItems(strList,isEditable);
ui->tableView->setItemDelegateForColumn(
    tabModel->fieldIndex("Department"), &delegateDepart);

//创建界面组件与数据模型的字段之间的数据映射
dataMapper= new QDataWidgetMapper();
dataMapper->setModel(tabModel); //设置数据模型
dataMapper->setSubmitPolicy(QDataWidgetMapper::AutoSubmit);
//界面组件与 tabModel 的具体字段之间的联系
dataMapper->addMapping(ui->dbSpinEmpNo, tabModel->fieldIndex("EmpNo"));
dataMapper->addMapping(ui->dbEditName, tabModel->fieldIndex("Name"));
dataMapper->addMapping(ui->dbComboSex, tabModel->fieldIndex("Gender"));
dataMapper->addMapping(ui->dbSpinHeight, tabModel->fieldIndex("Height"));
dataMapper->addMapping(ui->dbEditBirth, tabModel->fieldIndex("Birthday"));
dataMapper->addMapping(ui->dbEditMobile, tabModel->fieldIndex("Mobile"));
dataMapper->addMapping(ui->dbComboProvince,
    tabModel->fieldIndex("Province"));
dataMapper->addMapping(ui->dbEditCity, tabModel->fieldIndex("City"));
dataMapper->addMapping(ui->dbComboDep,
    tabModel->fieldIndex("Department"));
dataMapper->addMapping(ui->dbComboEdu, tabModel->fieldIndex("Education"));
dataMapper->addMapping(ui->dbSpinSalary, tabModel->fieldIndex("Salary"));
dataMapper->addMapping(ui->dbEditMemo, tabModel->fieldIndex("Memo"));
dataMapper->toFirst(); //移动到首记录

getFieldNames(); //获取字段名称列表, 填充 ui->groupBoxSort 组件
//更新 actions 和界面组件的使能状态
ui->actOpenDB->setEnabled(false);
ui->actRecAppend->setEnabled(true);
ui->actRecInsert->setEnabled(true);
ui->actRecDelete->setEnabled(true);
ui->actScan->setEnabled(true);
ui->groupBoxSort->setEnabled(true);
ui->groupBoxFilter->setEnabled(true);
}

```

函数 `openTable()` 主要是创建 `QSqlTableModel` 类型的私有变量 `tabModel`, 指定需要打开的数据表为 `employee`, 设置与界面显示组件的关联等。`QSqlTableModel` 设置一个数据表名称后, 作为数据表的数据模型, 可以方便地编辑一个数据表。

使用的核心的类是 `QSqlTableModel`, 其主要的函数功能见表 11-7 (省略了函数中的 `const` 关键字, 省略了缺省参数)。



表 11-7 QSqlTableModel 类的主要函数

函数	功能描述
QSqlDatabase database()	返回其数据库连接
void setTable(QString &tableName)	设置数据表名称，不立即读取记录，但会提取字段信息
QString tableName()	返回设置的数据表名称
void setFilter(QString &filter)	设置记录过滤条件
void setSort(int column, Qt::SortOrder order)	设置排序字段和排序规则，需调用 select()才生效
void sort(int column, Qt::SortOrder order)	按列号 column 和排序规则立即进行排序并获取数据
void setEditStrategy(EditStrategy strategy)	设置编辑策略
bool setHeaderData(int section, Qt::Orientation orientation, QVariant &value,)	设置表头，一般用于设置字段的显示名称
bool isDirty()	若有未更新到数据库的修改，就返回 true；否则返回 false
int fieldIndex(QString &fieldName)	根据字段名称返回其在模型中的字段序号，若字段不存在则返回-1
QSqlIndex primaryKey()	返回数据表的主索引
int rowCount()	返回记录条数
bool select()	查询数据表的数据，并使用设置的排序和过滤规则
bool selectRow(int row)	刷新获取指定行号的记录
void clear()	清除数据模型，释放所有获取的数据
QSqlRecord record()	返回一条空记录，只有字段名，可用来获取字段信息
QSqlRecord record(int row)	返回行号为 row 的一条记录，包含记录的数据
bool setRecord(int row, QSqlRecord &values)	更新一条记录的数据到数据模型，源和目标之间通过字段名称匹配，而不是按位置匹配
bool insertRecord(int row, QSqlRecord &record)	在行号 row 之前插入一条记录
bool insertRows(int row, int count)	在行号 row 之前插入 count 空行，编辑策略为 OnFieldChange 或 OnRowChange 时只能插入一行
bool removeRows(int row, int count)	从行号 row 开始，删除 count 行。若编辑策略为 OnManualSubmit，需调用 submitAll()才从数据表里删除
void revertRow(int row)	取消行号为 row 的记录的修改
void revert()	编辑策略为 OnRowChange 或 OnFieldChange 时，取消当前行的修改，对 OnManualSubmit 编辑策略无效
void revertAll()	取消所有未提交的修改
bool submit()	提交当前行的修改到数据库，对 OnManualSubmit 编辑策略无效
bool submitAll()	提交所有未更新的修改到数据库，若成功则返回 true；否则返回 false，错误的详细信息可以从 lastError()获取

函数 openTable()的代码主要包括以下几个部分的功能。

### 1. 数据模型创建与属性设置

首先创建 QSqlTableModel 类型的私有变量 tabModel，并且在创建时指定了数据库连接，就是前面设置的私有变量 DB，然后用 setTable()函数指定数据表。

setSort()函数设置排序字段和排序方式，其函数原型为：

```
void QSqlTableModel::setSort(int column, Qt::SortOrder order)
```

第 1 个参数 column 是排序字段的列号；第 2 个参数 order 是枚举类型 Qt::SortOrder，表示排序方式，取值 Qt::AscendingOrder 表示升序，Qt::DescendingOrder 表示降序。程序中设置为根据 EmpNo 字段升序排序，即：

```
tabModel->setSort(tabModel->fieldIndex("EmpNo"),Qt::AscendingOrder);
```

QSqlTableModel::setEditStrategy()函数用于设置编辑策略，参数是枚举类型 QSqlTableModel::EditStrategy，各取值的意义如下。

- QSqlTableModel::OnFieldChange, 字段值变化时立即更新到数据库。
- QSqlTableModel::OnRowChange, 当前行(就是记录)变换时更新到数据库。
- QSqlTableModel::OnManualSubmit, 所有修改暂时缓存, 手动调用 submitAll() 保存所有修改, 或调用 revertAll() 函数取消所有未保存修改。

程序里设置编辑策略为 QSqlTableModel::OnManualSubmit, 在修改数据后并不直接提交更新, 只是让工具栏上的“保存”和“取消”按钮可用, 用户手动提交或取消修改。

对 tabModel 设置这些属性后, 调用 select() 函数打开数据表, 如果打开失败, 可以通过 QSqlTableModel::lastError() 函数获取上一错误信息的文本说明。

## 2. 表头设置

QSqlTableModel::setHeaderData() 函数用于设置每个字段的表头数据, 主要是设置显示标题。如果不进行表头设置, 在 TableView 里显示时将显示字段名作为表头。为此, 将每个字段的显示设置为相应的中文标题。例如, 设置“Name”字段的显示标题为“姓名”的代码是:

```
tabModel->setHeaderData(tabModel->fieldIndex("Name"), Qt::Horizontal, "姓名");
```

setHeaderData() 函数的第 1 个参数是字段的序号, 通过 fieldIndex() 函数可以获取某个字段名在数据模型里的序号, 避免直接使用数字时不便于理解和修改的问题。

## 3. 选择模型及其信号的作用

为数据模型创建一个选择模型的代码如下:

```
theSelection=new QItemSelectionModel(tabModel);
```

选择模型的作用是当用户在 TableView 组件上操作时, 获取当前选择的行、列信息, 并且在选择的单元格变化时发射 currentChanged() 信号, 在当前行变化时发射 currentRowChanged() 信号。

为 currentChanged() 信号编写槽函数 on\_currentChanged(), 获取 tabModel->isDirty() 函数的值, 根据是否有未提交的修改, 更新工具栏按钮“保存”或“修改”的使能状态。槽函数 on\_currentChanged() 的代码如下:

```
void MainWindow::on_currentChanged(const QModelIndex &current, const QModelIndex &previous)
{
    //更新 actSubmit 和 actRevert 的状态
    Q_UNUSED(current);
    Q_UNUSED(previous);
    ui->actSubmit->setEnabled(tabModel->isDirty()); //有未保存修改时可用
    ui->actRevert->setEnabled(tabModel->isDirty());
}
```

为 currentRowChanged() 信号编写槽函数 on\_currentRowChanged(), 用于在当前行变化时, 从新的记录里提取 Photo 字段的内容, 并将图片在 QLabel 组件中显示出来。on\_currentRowChanged() 的代码如下:

```
void MainWindow::on_currentRowChanged(const QModelIndex &current, const QModelIndex &previous)
{
    Q_UNUSED(previous);
    // 行切换时的状态控制
    ui->actRecDelete->setEnabled(current.isValid());
    ui->actPhoto->setEnabled(current.isValid());
    ui->actPhotoClear->setEnabled(current.isValid());
}
```

```

if (!current.isValid())
{
    ui->dbLabPhoto->clear(); //清除图片显示
    return;
}

dataMapper->setCurrentIndex(current.row()); //更新数据映射的行号
int curRecNo=current.row();//获取行号
QSqlRecord curRec=tabModel->record(curRecNo); //获取当前记录
if (curRec.isNull("Photo")) //图片字段内容为空
    ui->dbLabPhoto->clear();
else
{
    QByteArray data=curRec.value("Photo").toByteArray();
    QPixmap pic;
    pic.loadFromData(data);
    ui->dbLabPhoto->setPixmap(pic.scaledToWidth(
        ui->dbLabPhoto->size().width()));
}
}

```

槽函数 `on_currentRowChanged()` 传递的参数 `current` 是行切换后新的当前行的模型索引，首先根据 `current` 是否有效，更新 3 个 Action 的使能状态。若 `current` 是有效的，更新数据映射 `dataMapper` 的当前行，即：

```
dataMapper->setCurrentIndex(current.row());
```

这将使界面上的编辑框、下拉列表框等与字段关联的界面组件显示当前记录的内容。

由于没有现成的界面组件可以通过数据映射显示 BLOB 字段内的图片，在此槽函数里通过编程获取 Photo 字段的数据，并将其显示为一个 QLabel 组件的 pixmap。

用下面两行代码获取当前行的记录：

```

int curRecNo=current.row();//获取行号
QSqlRecord curRec=tabModel->record(curRecNo); //获取当前记录

```

`curRec` 是 `QSqlRecord` 类型，返回了当前记录的数据，可以获取当前记录的每个字段的内容。程序中它获取 Photo 字段的内容，并将其转换为图片后作为 `ui->dbLabPhoto` 组件的 pixmap 来显示。

#### 4. 表示记录的类 QSqlRecord

`QSqlRecord` 类记录了数据表的字段信息和一条记录的数据内容，`QSqlTableModel` 有两种参数的函数 `record()` 可以返回一条记录。

- `QSqlRecord QSqlTableModel::record()`，不带参数的 `record()` 函数，返回的一个 `QSqlRecord` 对象只有记录的字段定义，但是没有数据。这个函数一般用于获取一个数据表的字段定义。
- `QSqlRecord QSqlTableModel::record(int row)`，返回行号为 `row` 的记录，包括记录的字段定义和数据。

`QSqlRecord` 类封装了对记录的字段定义和数据的操作，其主要函数见表 11-8（省略了函数参数中的 `const` 关键字，对于具有不同参数的同名函数，只列出一种参数形式的函数）。



表 11-8 QSqlRecord 类的主要函数

函数原型	功能描述
<code>void clear()</code>	清除记录的所有字段定义和数据
<code>void clearValues()</code>	清除所有字段的数据，将字段数据内容设置为 null
<code>bool contains(QString &amp;name)</code>	判断记录是否含有名称为 name 的字段
<code>bool isEmpty()</code>	若记录里没有字段，返回 true；否则返回 false
<code>int count()</code>	返回记录的字段个数
<code>QString fieldName(int index)</code>	返回序号为 index 的字段的名称
<code>int indexOf(QString &amp;name)</code>	返回字段名称为 name 的字段的序号，如果字段不存在，返回 -1
<code>QSqlField field(QString &amp;name)</code>	返回字段名称为 name 的字段对象
<code>QVariant value(QString &amp;name)</code>	返回字段名称为 name 的字段的值
<code>void setValue(QString &amp;name, QVariant &amp;val)</code>	设置字段名称为 name 的字段的值为 val
<code>bool isNull(const QString &amp;name)</code>	判断字段名称为 name 的字段数据是否为 null
<code>void setNull(const QString &amp;name)</code>	设置名称为 name 的字段的值为 null

QSqlRecord 用于字段操作的函数一般有两种参数形式的同名函数，用字段序号或字段名表示一个字段，如 value() 函数返回一个字段的值，有如下两种参数形式的函数：

- `QVariant value(int index)`，返回序号为 index 的字段的值；
- `QVariant value(QString &name)`，返回字段名称为 name 的字段的值。

### 5. 表示字段的类 QSqlField

QSqlRecord 的 field() 函数返回某个字段，返回数据类型是 QSqlField。QSqlField 封装了字段定义信息和数据。字段的定义一般在设计数据表时就固定了，不用在 QSqlRecord 里修改。QSqlRecord 用于字段数据读写的函数见表 11-9（省略了函数参数中的 const 关键字）。

表 11-9 QSqlField 类的主要函数

接口函数	功能描述
<code>void clear()</code>	清除字段数据，设置为 NULL。如果字段是只读的，则不清除
<code>bool isNull()</code>	判断字段值是否为 NULL
<code>bool setReadOnly(bool readOnly)</code>	设置一个字段为只读，只读的字段不能用 setValue() 函数设置值，也不能用 clear() 函数清除值
<code>QVariant value()</code>	返回字段的值
<code>void setValue(QVariant &amp;value)</code>	设置字段的值

### 6. tableView 的设置

界面上用一个 QTableView 组件显示 tabModel 的表格数据内容，设置其数据模型和选择模型，并且将 Memo 和 Photo 两个字段的列设置为隐藏，因为在表格里难以显示备注文字和图片。

为在 tableView 中显示和编辑“性别”和“部门”两个字段，设计了基于 QComboBox 的自定义代理组件类 QWComboBoxDelegate。自定义代理组件 QWComboBoxDelegate 的设计方法在 5.5 节有详细介绍，这里只是稍作改变，增加了一个 setItems() 函数，用于初始化下拉列表框和设置是否可以编辑，这样一个 QWComboBoxDelegate 可以创建多个代理组件实例，分别用于“性别”和“部门”两个字段。

下面是相对于 5.5 节的 QWComboBoxDelegate 修改或增加的部分的代码，其余相同的部分没有列出。

```
class QWComboBoxDelegate : public QStyledItemDelegate
```



```

{
    Q_OBJECT
private:
    QStringList m_ItemList; //选择列表
    bool m_isEdit; //是否可编辑
public:
    void setItems(QStringList items, bool isEdit); //初始化设置列表内容
    QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem &option, const
QModelIndex &index) const Q_DECL_OVERRIDE;
};

```

下面是两个函数的实现代码:

```

void QComboBoxDelegate::setItems(QStringList items, bool isEdit)
{
    m_ItemList=items;
    m_isEdit=isEdit;
}
QWidget *QComboBoxDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem &option, const QModelIndex &index) const
{
    Q_UNUSED(option);
    Q_UNUSED(index);
    QComboBox *editor = new QComboBox(parent);
    for (int i=0;i<m_ItemList.count();i++) //从字符串列表初始下拉列表
        editor->addItem(m_ItemList.at(i));
    editor->setEditable(m_isEdit); //是否可编辑
    return editor;
}

```

## 7. 数据映射

QDataWidgetMapper 用于建立界面组件与数据模型之间的映射, 可以将界面的 QLineEdit、QCombobox 等组件与数据模型的一个字段关联起来。

创建 QDataWidgetMapper 类的变量 dataMapper 后, 用 setModel() 设置关联的数据模型, setSubmitPolicy() 函数设置数据提交策略, 有自动 (AutoSubmit) 和手动 (ManualSubmit) 两种方式。addMapping() 用于设置界面组件与数据模型的列的映射, 程序在界面上的各编辑组件与数据表的各字段之间建立了映射关系。Memo 字段可以与一个 QPlainTextEdit 的组件映射, 但是 Photo 是一个 BLOB 字段, 没有组件可以直接显示其内容。

QDataWidgetMapper 还有 toFirst()、toPrevious()、toNext() 和 toLast() 4 个函数用于在记录间移动, setCurrentIndex() 和 setCurrentModelIndex() 可以直接移动到某一行记录, revert() 和 submit() 用于手工取消或提交当前记录的修改, 当提交策略设置为 QDataWidgetMapper::AutoSubmit 时, 行切换时将自动提交修改。

## 8. 获取数据表的所有字段名称

自定义函数 getFieldNames() 获取数据表的所有字段名称, 并填充到界面上的“排序字段”下拉列表框里, 用于选择排序字段。getFieldNames() 的实现代码如下:

```

void MainWindow::getFieldNames()
{ //获取所有字段名称
    QSqlRecord emptyRec=tabModel->record(); //获取空记录, 只有字段名
}

```

```

for (int i=0;i<emptyRec.count();i++)
    ui->comboFields->addItem(emptyRec.fieldName(i));
}

```

这里使用了 `QSqlTableModel::record()` 函数获取一个空的记录，包括数据表的所有字段信息。所有这些初始化完成后，更新窗口上相关 Actions 的使能状态。

## 11.2.4 添加、插入与删除记录

工具栏上有“添加”“插入”“删除”3 个按钮用于记录操作。`actRecAppend` 用于添加记录，其槽函数代码如下：

```

void MainWindow::on_actRecAppend_triggered()
{
    //添加记录
    tabModel->insertRow(tabModel->rowCount(),QModelIndex());
    QModelIndex curIndex=tabModel->index(tabModel->rowCount()-1,1);
    theSelection->clearSelection();//清空选择
    theSelection->setCurrentIndex(curIndex,QItemSelectionModel::Select);
    int currow=curIndex.row(); //获得当前行号
    tabModel->setData(tabModel->index(currow,0),2000+tabModel->rowCount());
    tabModel->setData(tabModel->index(currow,2),"男");
}

```

`QSqlTableModel::insertRow(int row)` 函数在数据模型的 `row` 行前面插入一行记录，如果 `row` 大于或等于数据模型的总行数，则在最后添加一行记录。

`actRecInsert` 实现在当前行的前面插入一行，其槽函数代码如下：

```

void MainWindow::on_actRecInsert_triggered()
{
    //插入记录
    QModelIndex curIndex=ui->tableView->currentIndex();
    tabModel->insertRow(curIndex.row(),QModelIndex());
    theSelection->clearSelection();//清除已有选择
    theSelection->setCurrentIndex(curIndex,QItemSelectionModel::Select);
}

```

`actRecDelete` 用于删除当前记录，其槽函数实现代码如下：

```

void MainWindow::on_actRecDelete_triggered()
{
    //删除当前记录
    QModelIndex curIndex=theSelection->currentIndex();//获取当前模型索引
    tabModel->removeRow(curIndex.row()); //删除一行
}

```

在插入或删除记录操作，未提交保存之前，`tableView` 的左侧表头会以标记表示记录编辑状态，“\*”表示新插入的记录，“!”表示删除的记录。在保存或取消修改后，这些标记就消失，删除的记录行也从 `tableView` 里删除。

## 11.2.5 保存与取消修改

在打开数据表初始化时，设置数据模型的编辑策略为 `OnManualSubmit`，即手动提交修改。当数据模型的数据被修改后，不管是直接修改字段值，还是插入或删除记录，在未提交修改前，

QSqlTableModel::isDirty()函数返回 true, 就是利用这个函数在自定义槽函数 on\_currentChanged()里修改 actSubmit 和 actRevert 两个 Action 的使能状态。

actSubmit 用于保存修改, actRevert 用于取消修改, 对应工具栏上的“保存”和“取消”两个按钮, 下面是这两个 Action 的槽函数的代码。

```
void MainWindow::on_actSubmit_triggered()
{
    //保存修改
    bool res=tabModel->submitAll();
    if (!res)
        QMessageBox::information(this, "消息", "数据保存错误, 错误信息\n"
        +tabModel->lastError().text(), QMessageBox::Ok, QMessageBox::NoButton);
    else
    {
        ui->actSubmit->setEnabled(false);
        ui->actRevert->setEnabled(false);
    }
}

void MainWindow::on_actRevert_triggered()
{
    //取消修改
    tabModel->revertAll();
    ui->actSubmit->setEnabled(false);
    ui->actRevert->setEnabled(false);
}
}
```

QSqlTableModel::submitAll()函数用于将数据表所有未提交的修改保存到数据库, revertAll()函数取消所有修改。

调用 submitAll()函数保存数据时如果失败, 可以通过 lastError()函数获取错误的具体信息, 例如, Name 是必填字段, 添加记录时若没有填写 Name 字段的内容就提交, 就会出现错误信息提示对话框。

## 11.2.6 设置和清除照片

employee 数据表的 Photo 字段是 BLOB 字段, 用于存储图片文件。Photo 字段内容的显示已经在自定义槽函数 on\_currentRowChanged()里实现了, 就是在当前记录变化时提取 Photo 字段的内容, 并显示为 dbLabPhoto 的 pixmap。

actPhoto 和 actPhotoClear 是设置照片和清除照片的两个 Action, 与工具栏上的“设置照片”和“清除照片”按钮关联。actPhoto 的槽函数代码如下:

```
void MainWindow::on_actPhoto_triggered()
{
    //设置照片
    QString aFile=QFileDialog::getOpenFileName(this, "选择图片", "", "照片 (*.jpg)");
    if (aFile.isEmpty())
        return;

    QByteArray data;
    QFile* file=new QFile(aFile);
    file->open(QIODevice::ReadOnly);
    data = file->readAll();
    file->close();
}
```

```

int curRecNo=theSelection->currentIndex().row();
QSqlRecord curRec=tabModel->record(curRecNo); //获取当前记录
curRec.setValue("Photo",data); //设置字段数据
tabModel->setRecord(curRecNo,curRec);
QPixmap pic;
pic.load(aFile); //在界面上显示
ui->dbLabPhoto->setPixmap(pic.scaledToWidth(ui->dbLabPhoto->width()));
}

```

代码功能是用文件对话框选择一个图片文件，读取文件内容到 `QByteArray` 类型的变量 `data` 里，获取当前记录到变量 `curRec` 后，用 `QSqlRecord` 的 `setValue()` 函数为 `Photo` 字段设置数据为 `data`，然后用 `setRecord()` 函数更新当前记录。

```

curRec.setValue("Photo",data); //设置字段数据
tabModel->setRecord(curRecNo,curRec);

```

这里的更新只是更新到数据模型，并没有更新到数据库。

`actPhotoClear` 用于清除照片，其槽函数实现代码如下：

```

void MainWindow::on_actPhotoClear_triggered()
{
    int curRecNo=theSelection->currentIndex().row();
    QSqlRecord curRec=tabModel->record(curRecNo); //获取当前记录
    curRec.setNull("Photo");//设置为空值
    tabModel->setRecord(curRecNo,curRec);
    ui->dbLabPhoto->clear();
}

```

获取当前记录到变量 `curRec` 后，调用 `setNull()` 函数将 `Photo` 字段设置为 `NULL`，就是清除了字段的内容，然后更新记录到数据模型。

## 11.2.7 数据记录的遍历

工具栏上的“涨工资”按钮用于将数据表内所有记录的 `salary` 字段的内容增加 10%，演示了记录遍历的功能。`actScan` 实现此按钮的功能，其槽函数实现代码如下：

```

void MainWindow::on_actScan_triggered()
{ //涨工资，记录遍历
    if (tabModel->rowCount()==0)
        return;
    for (int i=0;i<tabModel->rowCount();i++)
    {
        QSqlRecord aRec=tabModel->record(i); //获取当前记录
        float salary=aRec.value("Salary").toFloat();
        salary=salary*1.1;
        aRec.setValue("Salary",salary);
        tabModel->setRecord(i,aRec);
    }
    if (tabModel->submitAll())
        QMessageBox::information(this, "消息", "涨工资计算完毕",
                                   QMessageBox::Ok,QMessageBox::NoButton);
}

```



## 11.2.8 记录排序

QSqlTableModel 的 `setSort()` 函数设置数据表根据某个字段按照升序或降序排列，实际上就是设置了 SQL 语句里的 ORDER BY 子句。

在打开数据库时，已经调用 `getFieldNames()` 函数将数据表的所有字段名添加到界面上的 `comboFields` 下拉列表框了。“排序字段”下拉列表框、“升序”和“降序”两个 `RadioButton` 的相应操作的槽函数实现按选择字段排序。

```
void MainWindow::on_comboFields_currentIndexChanged(int index)
{ //选择字段进行排序
    if (ui->radioBtnAscend->isChecked())
        tabModel->setSort(index, Qt::AscendingOrder);
    else
        tabModel->setSort(index, Qt::DescendingOrder);
    tabModel->select();
}
void MainWindow::on_radioBtnAscend_clicked()
{ //升序
    tabModel->setSort(ui->comboFields->currentIndex(), Qt::AscendingOrder);
    tabModel->select();
}
void MainWindow::on_radioBtnDescend_clicked()
{ //降序
    tabModel->setSort(ui->comboFields->currentIndex(), Qt::DescendingOrder);
    tabModel->select();
}
```

在调用 `setSort()` 函数设置排序规则后，需要调用 `QSqlTableModel::select()` 重新读取数据表的数据才会使排序规则生效。

## 11.2.9 记录过滤

QSqlTableModel 的 `setFilter()` 函数设置记录过滤条件，实际上就是设置了 SQL 语句里的 WHERE 字句。

实例里演示针对 `Gender` 字段设置字段过滤条件，窗口界面上“数据过滤”分组框里有 3 个 `RadioButton`，分别为“男”“女”“全显示”，3 个按钮的 `clicked()` 信号的槽函数实现代码如下：

```
void MainWindow::on_radioBtnMan_clicked()
{
    tabModel->setFilter(" Gender='男' ");
}
void MainWindow::on_radioBtnWoman_clicked()
{
    tabModel->setFilter(" Gender='女' ");
}
void MainWindow::on_radioBtnBoth_clicked()
{
    tabModel->setFilter("");
}
```

调用 `setFilter()` 后无需调用 `select()` 函数就可以立即刷新记录，若要取消过滤条件，只需在

setFilter()函数里传递一个空字符串。

## 11.3 QSqlQueryModel 的使用

### 11.3.1 QSqlQueryModel 功能概述

从图 11-2 的 SQL 数据模型类的继承关系看, QSqlQueryModel 是 QSqlTableModel 的父类。QSqlQueryModel 封装了执行 SELECT 语句从数据库查询数据的功能, 但是 QSqlQueryModel 只能作为只读数据源使用, 不可以编辑数据。

QSqlQueryModel 类的主要接口函数见表 11-10 (省略了函数中的 const 关键字和缺省参数)。

表 11-10 QSqlQueryModel 类的主要函数

接口函数	功能描述
void clear()	清除数据模型, 释放所有获得的数据
QSqlError lastError()	返回上次的错误, 可获取错误的类型和文本信息
QSqlQuery query()	返回当前关联的 QSqlQuery 对象
void setQuery(QSqlQuery &query)	设置一个 QSqlQuery 对象, 获取数据
void setQuery(QString &query)	设置一个 SELECT 语句创建查询, 获取数据
QSqlRecord record()	返回一个空记录, 包含当前查询的字段信息
QSqlRecord record(int row)	返回行号为 row 的记录
int rowCount()	返回查询到的记录条数
int columnCount()	返回查询的字段个数
void setHeaderData(int section, Qt::Orientation orientation, QVariant &value)	设置表头数据, 一般用于设置字段的表头标题

使用 QSqlQueryModel 作为数据模型从数据库里查询数据, 只需使用 setQuery()函数设置一个 SELECT 查询语句即可。QSqlQueryModel 可以作为 QTableView 等视图组件的数据源, 也可以使用 QDataWidgetMapper 创建字段与界面组件的映射, 只是查询出来的数据是不可编辑的。

### 11.3.2 使用 QSqlQueryModel 实现数据查询

#### 1. 实例功能

使用 QSqlQueryModel 可以从一个数据表或多个数据表里查询数据, 只需设计好 SELECT 语句即可。实例 samp11\_2 使用 QSqlQueryModel 从 employee 表里查询记录, 并在界面上显示, 运行窗口见图 11-5。

窗口工具栏上几个工具栏按钮只有打开数据库和记录移动功能, 记录移动通过调用 QdataWidgetMapper 类的记录移动功能实现。窗口上没有数据编辑和保存功能, 因为 QSqlQueryModel 查询出来的数据是只读的。

主窗口类的定义部分如下 (去除了界面组件自动生成的槽函数的定义):

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
```

```

QSqlDatabase DB; //数据库
QSqlQueryModel *qryModel; //数据模型
QItemSelectionModel *theSelection; //选择模型
QDataWidgetMapper *dataMapper; //数据界面映射
void openTable(); //打开数据表
void refreshTableView(); //移动记录时刷新 TableView 的当前行
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    //selectionModel 的行发生了变化
    void on_currentRowChanged(const QModelIndex &current, const QModelIndex &previous);
};

```

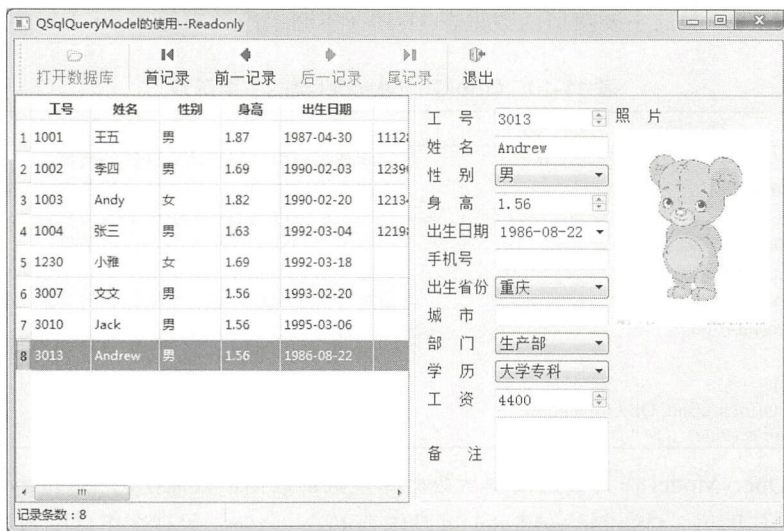


图 11-5 实例 samp11\_2 运行窗口

这里定义了 QSqlQueryModel 类型的数据模型变量 qryModel, 也定义了数据库、选择模型和数据界面映射的变量。refreshTableView()函数用于记录移动后刷新 tableView 上的当前行位置。

自定义槽函数 on\_currentRowChanged()在选择模型的当前行变化时, 处理 Photo 字段的查询与照片显示。

## 2. 打开数据库

工具栏上的“打开数据库”按钮对应 actOpenDB, 选择 SQLite 数据库文件, 然后调用 openTable() 函数打开数据库。槽函数 on\_actOpenDB\_triggered()的代码与实例 samp11\_1 完全一样, 这里不再列出。

openTable()用于查询数据, 建立界面显示等具体操作, 其代码如下:

```

void MainWindow::openTable()
{ //打开数据表
    qryModel=new QSqlQueryModel(this);
    qryModel->setQuery("SELECT EmpNo, Name, Gender, Height, Birthday, Mobile, Province,
City, Department , Education, Salary FROM employee ORDER BY EmpNo");
    if (qryModel->lastError().isValid())

```

```

{
    QMessageBox::critical(this, "错误",
        "数据表查询错误, 错误信息\n"+qryModel->lastError().text(),
        QMessageBox::Ok, QMessageBox::NoButton);
    return;
}

qryModel->setHeaderData(0, Qt::Horizontal, "工号");
qryModel->setHeaderData(1, Qt::Horizontal, "姓名");
qryModel->setHeaderData(2, Qt::Horizontal, "性别");
qryModel->setHeaderData(3, Qt::Horizontal, "身高");
qryModel->setHeaderData(4, Qt::Horizontal, "出生日期");
qryModel->setHeaderData(5, Qt::Horizontal, "手机");
qryModel->setHeaderData(6, Qt::Horizontal, "省份");
qryModel->setHeaderData(7, Qt::Horizontal, "城市");
qryModel->setHeaderData(8, Qt::Horizontal, "部门");
qryModel->setHeaderData(9, Qt::Horizontal, "学历");
qryModel->setHeaderData(10, Qt::Horizontal, "工资");

theSelection=new QItemSelectionModel(qryModel);
connect(theSelection, SIGNAL(currentRowChanged(QModelIndex, QModelIndex)),
    this, SLOT(on_currentRowChanged(QModelIndex, QModelIndex)));
ui->tableView->setModel(qryModel);
ui->tableView->setSelectionModel(theSelection);
//创建数据映射
dataMapper= new QDataWidgetMapper();
dataMapper->setSubmitPolicy(QDataWidgetMapper::AutoSubmit);
dataMapper->setModel(qryModel);
dataMapper->addMapping(ui->dbSpinEmpNo, 0); //"EmpNo";
dataMapper->addMapping(ui->dbEditName, 1); //"Name";
dataMapper->addMapping(ui->dbComboSex, 2); //"Gender";
dataMapper->addMapping(ui->dbSpinHeight, 3); //"Height";
dataMapper->addMapping(ui->dbEditBirth, 4); //"Birthday";
dataMapper->addMapping(ui->dbEditMobile, 5); //"Mobile";
dataMapper->addMapping(ui->dbComboProvince, 6); //"Province";
dataMapper->addMapping(ui->dbEditCity, 7); //"City";
dataMapper->addMapping(ui->dbComboDep, 8); //"Department";
dataMapper->addMapping(ui->dbComboEdu, 9); //"Education";
dataMapper->addMapping(ui->dbSpinSalary, 10); //"Salary";
dataMapper->toFirst();
ui->actOpenDB->setEnabled(false);
}

```

程序首先创建了 QSqlQueryModel 类型的私有变量 qryModel, 然后调用 setQuery() 函数设置了 SELECT 查询语句, SELECT 语句从 employee 表里查询除了 Memo 和 Photo 之外的所有其他字段。

使用 setHeaderData() 函数为每个字段设置显示标题, 为使代码简化, 这里直接使用了字段的序号。

为 qryModel 创建了选择模型 theSelection, 并且将其 currentRowChanged() 信号与自定义槽函数 on\_currentRowChanged() 关联起来。这个自定义槽函数用来在记录移动时, 查询出 Memo 和 Photo 字段的内容, 并在界面上显示出来。

下面是槽函数 on\_currentRowChanged() 的代码:

```
void MainWindow::on_currentRowChanged(const QModelIndex &current, const QModelIndex &previous)
```



```

{
    Q_UNUSED(previous);
    if (!current.isValid())
    {
        ui->dbLabPhoto->clear();
        return;
    }
    dataMapper->setCurrentModelIndex(current); //更新数据映射的行号
    bool first=(current.row()==0); //是否首记录
    bool last=(current.row()==qryModel->rowCount()-1); //是否尾记录
    ui->actRecFirst->setEnabled(!first); //更新使能状态
    ui->actRecPrevious->setEnabled(!first);
    ui->actRecNext->setEnabled(!last);
    ui->actRecLast->setEnabled(!last);

    int curRecNo=theSelection->currentIndex().row();
    QSqlRecord curRec=qryModel->record(curRecNo); //获取当前记录
    int empNo=curRec.value("EmpNo").toInt();
    QSqlQuery query; //查询当前 EmpNo 的 Memo 和 Photo 字段的数据
    query.prepare("select EmpNo, Memo, Photo from employee where EmpNo = :ID");
    query.bindValue(":ID", empNo);
    query.exec();
    query.first();

    QVariant va=query.value("Photo");
    if (!va.isValid()) //图片字段内容为空
        ui->dbLabPhoto->clear();
    else
    {
        //显示照片
        QByteArray data=va.toByteArray();
        QPixmap pic;
        pic.loadFromData(data);
        ui->dbLabPhoto->setPixmap(pic.scaledToWidth(
            ui->dbLabPhoto->size().width()));
    }

    QVariant va2=query.value("Memo"); //显示备注
    ui->dbEditMemo->setPlainText(va2.toString());
}

```

这个函数实现 3 个功能，第 1 个功能是更新数据映射的行号，即：

```
dataMapper->setCurrentModelIndex(current);
```

使窗口上的字段关联的显示组件刷新显示当前记录的内容。

第 2 个功能是根据当前行号，判断是否是首记录或尾记录，以此更新界面上 4 个记录移动的 Action 的使能状态。

第 3 个功能是获取当前记录的 EmpNo 字段的值（即员工编号），然后用一个 QSqlQuery 变量 query 执行查询语句，只查询出这个员工的 Memo 和 Photo 字段的数据，然后在界面元件上显示。这里使用了 QSqlQuery 类，它用来执行任意的 SQL 语句。

在 QSqlQueryModel 类的变量 qryModel 里设置 SELECT 语句时，并没有查询所有字段，因为 Photo 是 BLOB 字段，全部查询出来后必然占用较大内存，而且在做记录遍历时，如果存在 BLOB 字段数据，执行速度会很慢。所以，这个实例里将普通字段的查询用 QSqlQueryModel 来查询并显示，而 Memo 和 Photo 字段数据的查询采用按需查询的方式，这样可以减少内存消耗，提高记录遍历时的执行速度。

openTable()函数剩余的部分是设置 tableView 的数据模型和选择模型，然后创建数据界面映射变量 dataMapper，设置各个界面组件与字段的映射关系。

### 3. 记录移动

用于数据映射的 QDataWidgetMapper 类设置数据模型后，总是指向数据模型的当前记录。QDataWidgetMapper 有 4 个函数进行当前记录的移动，分别是 toFirst()、toLast()、toNext()和 toPrevious()。当前记录移动时，会引起数据模型关联的选择模型发射 currentRowChanged()信号，也就会执行关联的自定义槽函数 on\_currentRowChanged()。

工具栏上有 4 个记录移动的按钮，它们调用 QDataWidgetMapper 的记录移动函数实现记录移动，4 个 Action 的槽函数代码如下：

```
void MainWindow::on_actRecFirst_triggered()
{ //首记录
    dataMapper->toFirst();
    refreshTableView();
}
void MainWindow::on_actRecPrevious_triggered()
{ //前一条记录
    dataMapper->toPrevious();
    refreshTableView();
}
void MainWindow::on_actRecNext_triggered()
{ //后一条记录
    dataMapper->toNext();
    refreshTableView();
}
void MainWindow::on_actRecLast_triggered()
{ //最后一条记录
    dataMapper->toLast();
    refreshTableView();
}
```

使用 QDataWidgetMapper 的记录移动操作后，QDataWidgetMapper 会移动到新的记录上，映射了字段的界面组件也会自动显示新记录的字段的数据。但是，tableView 的当前行并不会自动变化，所以需要调用 refreshTableView()函数刷新 tableView 的显示，refreshTableView()函数的代码如下：

```
void MainWindow::refreshTableView()
{ //刷新 tableView 的当前选择行
    int index=dataMapper->currentIndex();
    QModelIndex curIndex=qryModel->index(index,1);
    theSelection->clearSelection(); //清空选择项
    theSelection->setCurrentIndex(curIndex,QItemSelectionModel::Select);
}
```

## 11.4 QSqlQuery 的使用

### 11.4.1 QSqlQuery 基本用法

QSqlQuery 是能执行任意 SQL 语句的类，如 SELECT、INSERT、UPDATE、DELETE 等，

QSqlQuery 类的一些常用函数见表 11-11（省略函数中的 const 关键字，省略缺省参数，不同参数的同名函数一般只给出一种参数形式）。

表 11-11 QSqlQuery 类的主要函数

接口函数	功能描述
bool prepare(QString &query)	设置准备执行的 SQL 语句，一般用于带参数的 SQL 语句
void bindValue(QString &placeholder, QVariant &val)	设置 SQL 语句中参数的值，以占位符表示参数
void exec()	执行由 prepare()和 bindValue()设置的 SQL 语句
void exec(QString &query)	直接执行一个不带参数的 SQL 语句
bool isActive()	如果成功执行了 exec()函数，就返回 true
bool isSelected()	如果执行的 SQL 语句是 SELECT 语句，就返回 true
QSqlRecord record()	返回当前记录
QVariant value(QString &name)	返回当前记录名称为 name 的字段的值
bool isNull(QString &name)	判断一个字段是否为空，当 query 非活动、未定位在有效记录、无此字段或字段为空时都返回 true
int size()	对于 SELECT 语句，返回查询到的记录条数，其他语句返回-1
int numRowsAffected()	返回 SQL 语句影响的记录条数，对于 SELECT 语句无定义
bool first()	定位到第一条记录，isActive 和 isSelected 都为 true 时才有效
bool previous()	定位到上一条记录，isActive 和 isSelected 都为 true 时才有效
bool next()	定位到下一条记录，isActive 和 isSelected 都为 true 时才有效
bool last()	定位到最后一条记录，isActive 和 isSelected 都为 true 时才有效
bool seek(int index)	定位到指定序号的记录
int at()	返回当前记录的序号

使用 QSqlQuery 执行不带参数的 SQL 语句时可以用 exec(QString)函数，如：

```
QSqlQuery query;
query.exec("SELECT * FROM employee");
query.exec("UPDATE employee SET Salary=3000 where Gender='女' ");
```

使用带参数的 SQL 语句时，先用 prepare()函数准备 SQL 语句，然后用 bindValue()函数设置参数值，再用 exec()执行 SQL 语句，如：

```
QSqlQuery query;
query.prepare ("SELECT * FROM employee where EmpNo=:ID");
query.bindValue(":ID", 2003);
query.exec();
```

上面是 SQL 语句中的参数用“冒号+参数名”表示的形式，还可以直接用占位符来表示参数，如：

```
QSqlQuery query;
query.prepare ("UPDATE employee SET Name=?, Gender=?, Height=? where EmpNo=?");
query.bindValue(0, "高某某");
query.bindValue(1, "男");
query.bindValue(2, 1.78);
query.bindValue(3, 2010);
query.exec();
```

## 11.4.2 QSqlQueryModel 和 QSqlQuery 联合使用

### 1. 数据表显示

QSqlQueryModel 可以查询数据并作为数据模型，实现数据的显示，QSqlQuery 可以执行

UPDATE、INSERT、DELETE 等 SQL 语句实现数据的编辑修改。

实例 samp11\_3 通过联合使用 QSqlQueryModel 和 QSqlQuery 组件实现数据表的显示和编辑修改，图 11-6 是实例 samp11\_3 运行主窗口。



图 11-6 实例 samp11\_3 运行窗口

主窗口类定义如下（去掉了一些自动生成的部分）：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QSqlDatabase DB; //数据库
    QSqlQueryModel *qryModel; //数据库模型
    QItemSelectionModel *theSelection; //选择模型
    void openTable(); //打开数据表
    void updateRecord(int recNo); //更新记录
public:
    explicit MainWindow(QWidget *parent = 0);
};
```

这个主窗口类的定义相比于实例 samp11\_2 只是增加了一个 updateRecord() 函数。工具栏上的“打开数据库”按钮的代码与实例 samp11\_2 完全相同，会调用 openTable() 连接数据库并查询数据表的数据。

本实例的 openTable() 函数的代码如下：

```
void MainWindow::openTable()
{ //打开数据表
    qryModel=new QSqlQueryModel(this);
    theSelection=new QItemSelectionModel(qryModel);
    qryModel->setQuery("SELECT EmpNo, Name, Gender, Height, Birthday, Mobile, Province,
City, Department, Education, Salary FROM employee order by EmpNo");
    if (qryModel->lastError().isValid())
    {
        QMessageBox::information(this, "错误", "数据表查询错误, 错误信息\n"
+qryModel->lastError().text(), QMessageBox::Ok, QMessageBox::NoButton);
        return;
    }
    qryModel->setHeaderData(0,Qt::Horizontal,"工号");
    qryModel->setHeaderData(1,Qt::Horizontal,"姓名");
```



```

qryModel->setHeaderData(2,Qt::Horizontal,"性别");
qryModel->setHeaderData(3,Qt::Horizontal,"身高");
qryModel->setHeaderData(4,Qt::Horizontal,"出生日期");
qryModel->setHeaderData(5,Qt::Horizontal,"手机");
qryModel->setHeaderData(6,Qt::Horizontal,"省份");
qryModel->setHeaderData(7,Qt::Horizontal,"城市");
qryModel->setHeaderData(8,Qt::Horizontal,"部门");
qryModel->setHeaderData(9,Qt::Horizontal,"学历");
qryModel->setHeaderData(10,Qt::Horizontal,"工资");

ui->tableView->setModel(qryModel);
ui->tableView->setSelectionModel(theSelection);
ui->actOpenDB->setEnabled(false);
ui->actRecInsert->setEnabled(true);
ui->actRecDelete->setEnabled(true);
ui->actRecEdit->setEnabled(true);
ui->actScan->setEnabled(true);
}

```

openTable()函数创建了 QSqlQueryModel 类对象 qryModel,从数据表 employee 里查询出除了 Memo 和 Photo 之外的其他字段,并作为界面上的 tableView 的数据模型。还创建了选择模型 theSelection,但是没有为选择模型的 currentRowChanged()信号关联槽函数,因为不需要在记录移动时做什么处理。由于使用 QSqlQueryModel 作为 tableView 的数据源,在 tableView 里是无法编辑修改数据的。

## 2. 编辑记录对话框

由于在 tableView 上无法编辑修改数据,只是作为一个只读的数据显示,在主窗口工具栏上提供了“插入记录”“编辑记录”“删除记录”3个按钮对数据进行编辑。“插入记录”和“编辑记录”都会打开一个对话框,编辑一条记录的所有字段数据,确认插入后用 QSqlQuery 执行一条 INSERT 语句插入一条记录,确认编辑时用 QSqlQuery 执行一个 UPDATE 语句更新一条记录。

设计了一个对话框 WDialogData,用于编辑一条记录的所有字段的数据,在“插入记录”和“编辑记录”时调用此对话框,对话框运行界面如图 11-7 所示。



图 11-7 设计时的 WDialogData 对话框界面

WDialogData 类的定义如下:

```
class WDialogData : public QDialog
{
    Q_OBJECT
private:
    QSqlRecord mRecord; //保存一条记录的数据
public:
    explicit WDialogData(QWidget *parent = 0);
    ~WDialogData();
    void setUpdateRecord(QSqlRecord &recData); //更新记录
    void setInsertRecord(QSqlRecord &recData); //插入记录
    QSqlRecord getRecordData(); //获取录入的数据
private slots:
    void on_btnClearPhoto_clicked(); //清理照片
    void on_btnSetPhoto_clicked(); //设置照片
private:
    Ui::WDialogData *ui;
};
```

QSqlRecord 类型的私有变量 mRecord 用于存储一条记录的数据。

插入一条记录时, 创建对话框后调用 setInsertRecord() 函数初始化对话框的数据; 编辑一条记录时, 创建对话框后调用 setUpdateRecord() 函数初始化对话框的数据。

对话框确认修改后, 调用 getRecordData() 时, 界面数据存入 mRecord, 并将 mRecord 作为返回值, 返回编辑后的一条记录的数据。

对话框 WDialogData 的所有自定义函数, 以及对话框上的“导入照片”“清除照片”按钮的代码如下, 程序代码比较简单, 这里不再过多解释。

```
void WDialogData::setUpdateRecord(QSqlRecord &recData)
{ //编辑记录, 更新记录数据到界面
    mRecord=recData;
    ui->spinEmpNo->setEnabled(false); //员工编号不允许编辑
    setWindowTitle("更新记录");
    //根据 recData 的数据更新界面显示
    ui->spinEmpNo->setValue(recData.value("EmpNo").toInt());
    ui->editName->setText(recData.value("Name").toString());
    ui->comboSex->setCurrentText(recData.value("Gender").toString());
    ui->spinHeight->setValue(recData.value("Height").toFloat());
    ui->editBirth->setDate(recData.value("Birthday").toDate());
    ui->editMobile->setText(recData.value("Mobile").toString());
    ui->comboProvince->setCurrentText(recData.value("Province").toString());
    ui->editCity->setText(recData.value("City").toString());
    ui->comboDep->setCurrentText(recData.value("Department").toString());
    ui->comboEdu->setCurrentText(recData.value("Education").toString());
    ui->spinSalary->setValue(recData.value("Salary").toInt());
    ui->editMemo->setPlainText(recData.value("Memo").toString());
    QVariant va=recData.value("Photo");
    if (!va.isValid()) //图片字段内容为空
        ui->LabPhoto->clear();
    else
    {
        QByteArray data=va.toByteArray();
        QPixmap pic;
```

```

        pic.loadFromData(data);
        ui->LabPhoto->setPixmap(pic.scaledToWidth(
            ui->LabPhoto->size().width()));
    }
}

void WDialogData::setInsertRecord(QSqlRecord &recData)
{ // 插入记录, 无需更新界面显示, 但是要存储 recData 的字段结构
    mRecord=recData; // 保存 recData 到内部变量
    ui->spinEmpNo->setEnabled(true); // 插入的记录, 员工编号允许编辑
    setWindowTitle("插入新记录");
    ui->spinEmpNo->setValue(recData.value("EmpNo").toInt());
}

QSqlRecord WDialogData::getRecordData()
{ // "确定"按钮后, 界面数据保存到记录 mRecord
    mRecord.setValue("EmpNo", ui->spinEmpNo->value());
    mRecord.setValue("Name", ui->editName->text());
    mRecord.setValue("Gender", ui->comboSex->currentText());
    mRecord.setValue("Height", ui->spinHeight->value());
    mRecord.setValue("Birthday", ui->editBirth->date());
    mRecord.setValue("Mobile", ui->editMobile->text());
    mRecord.setValue("Province", ui->comboProvince->currentText());
    mRecord.setValue("City", ui->editCity->text());
    mRecord.setValue("Department", ui->comboDep->currentText());
    mRecord.setValue("Education", ui->comboEdu->currentText());
    mRecord.setValue("Salary", ui->spinSalary->value());
    mRecord.setValue("Memo", ui->editMemo->toPlainText());
    // 照片编辑时已经修改了 mRecord 的 Photo 字段的值
    return mRecord; // 以记录作为返回值
}

void WDialogData::on_btnClearPhoto_clicked()
{ // 清除照片
    ui->LabPhoto->clear();
    mRecord.setNull("Photo"); // Photo 字段清空
}

void WDialogData::on_btnSetPhoto_clicked()
{ // 设置照片
    QString aFile=QFileDialog::getOpenFileName(this, "选择图片", "", "照片(*.jpg)");
    if (aFile.isEmpty())
        return;
    QByteArray data;
    QFile* file=new QFile(aFile);
    file->open(QIODevice::ReadOnly);
    data = file->readAll();
    file->close();
    mRecord.setValue("Photo", data); // 图片保存到 Photo 字段
    QPixmap pic;
    pic.loadFromData(data);
    ui->LabPhoto->setPixmap(pic.scaledToWidth(ui->LabPhoto->size().width()));
}

```

### 3. 编辑记录

单击主窗口工具栏上的“编辑记录”按钮, 或在 tableView 上双击某条记录, 会编辑当前记录, 代码如下:

```

void MainWindow::on_actRecEdit_triggered()
{ //编辑当前记录
    int curRecNo=theSelection->currentIndex().row();
    updateRecord(curRecNo);
}
void MainWindow::on_tableView_doubleClicked(const QModelIndex &index)
{ //tableView 上双击, 编辑当前记录
    int curRecNo=index.row();
    updateRecord(curRecNo);
}

```

两个槽函数都会调用 updateRecord() 函数, 并且以记录的序号作为传递参数。updateRecord() 函数实现当前记录的编辑, 代码如下:

```

void MainWindow::updateRecord(int recNo)
{ //更新一条记录
    QSqlRecord curRec=qryModel->record(recNo); //获取当前记录
    int empNo=curRec.value("EmpNo").toInt(); //获取 EmpNo
    QSqlQuery query; //查询出当前记录的所有字段
    query.prepare("select * from employee where EmpNo = :ID");
    query.bindValue(":ID", empNo);
    query.exec();
    query.first();
    if (!query.isValid()) //是否为有效记录
        return;

    curRec=query.record();
    WDialogData *dataDialog=new WDialogData(this);
    Qt::WindowFlags flags=dataDialog->windowFlags();
    dataDialog->setWindowFlags(flags | Qt::MSWindowsFixedSizeDialogHint);
    dataDialog->setUpdateRecord(curRec); //调用对话框函数更新数据和界面
    int ret=dataDialog->exec(); //以模式方式显示对话框
    if (ret==QDialog::Accepted) //OK 键被按下
    {
        QSqlRecord recData=dataDialog->getRecordData(); //获得对话框返回的记录
        query.prepare("update employee set Name=:Name, Gender=:Gender, "
" Height=:Height, Birthday=:Birthday, Mobile=:Mobile, Province=:Province, "
" City=:City, Department=:Department, Education=:Education, Salary=:Salary, "
" Memo=:Memo, Photo=:Photo where EmpNo = :ID");
        query.bindValue(":Name", recData.value("Name"));
        query.bindValue(":Gender", recData.value("Gender"));
        query.bindValue(":Height", recData.value("Height"));
        query.bindValue(":Birthday", recData.value("Birthday"));
        query.bindValue(":Mobile", recData.value("Mobile"));
        query.bindValue(":Province", recData.value("Province"));
        query.bindValue(":City", recData.value("City"));
        query.bindValue(":Department", recData.value("Department"));
        query.bindValue(":Education", recData.value("Education"));
        query.bindValue(":Salary", recData.value("Salary"));
        query.bindValue(":Memo", recData.value("Memo"));
        query.bindValue(":Photo", recData.value("Photo"));
        query.bindValue(":ID", empNo);
        if (!query.exec())
            QMessageBox::critical(this, "错误",
                "记录更新错误\n"+query.lastError().text(),
                QMessageBox::Ok, QMessageBox::NoButton);
    }
}

```



```

        else//重新执行 SQL 语句查询
            qryModel->query().exec();
    }
    delete dataDialog;
}

```

函数 `updateRecord(int recNo)` 根据行号 `recNo` 从 `qryModel` 获取当前记录的 `EmpNo` 字段的值, 即员工编号, 然后使用一个 `QSqlQuery` 对象从数据表里查询出这一个员工的所有字段的一条记录。使用 `QsqlQuery` 时用到了参数 SQL 语句:

```

query.prepare("select * from employee where EmpNo = :ID");
query.bindValue(":ID", empNo);
query.exec();

```

由于 `EmpNo` 是数据表 `employee` 的主键字段, 不允许出现重复, 所以只会查询出一条记录, 查询出的一条完整记录保存到变量 `curRec`。

然后创建 `WDialogData` 类型的对话框 `dataDialog`, 调用 `setUpdateRecord(curRec)` 将完整记录传递给对话框。对话框执行后, 如果是“确定”返回, 则通过 `getRecordData()` 函数获取对话框编辑后的记录数据:

```

QSqlRecord recData=dataDialog->getRecordData();

```

`recData` 里包含了编辑后的最新数据, 然后使用 `QSqlQuery` 对象执行带参数的 `UPDATE` 语句更新一条记录。更新成功后, 将数据模型 `qryModel` 的 `SELECT` 语句重新执行一次, 可刷新 `tableView` 的显示。

#### 4. 插入记录

工具栏上的“插入记录”可以插入一条新记录, 代码如下:

```

void MainWindow::on_actRecInsert_triggered()
{
    //插入记录
    QSqlQuery query;
    query.exec("select * from employee where EmpNo =-1"); //只查询字段信息
    QSqlRecord curRec=query.record(); //获取当前记录, 实际为空记录
    curRec.setValue("EmpNo", qryModel->rowCount()+3000);

    WDialogData *dataDialog=new WDialogData(this);
    Qt::WindowFlags flags=dataDialog->windowFlags();
    dataDialog->setWindowFlags(flags | Qt::MSWindowsFixedSizeDialogHint);
    dataDialog->setInsertRecord(curRec); //插入记录
    int ret=dataDialog->exec(); //以模态方式显示对话框
    if (ret==QDialog::Accepted) //OK 键被按下
    {
        QSqlRecord recData=dataDialog->getRecordData();
        query.prepare("INSERT INTO employee (EmpNo,Name,Gender, Height,Birthday, "
"Mobile,Province, City,Department,Education,Salary,Memo,Photo) "
" VALUES (:EmpNo,:Name, :Gender,:Height,:Birthday, :Mobile, :Province,"
" :City, :Department, :Education,:Salary,:Memo,:Photo)");
        query.bindValue(":EmpNo", recData.value("EmpNo"));
        query.bindValue(":Name", recData.value("Name"));
        query.bindValue(":Gender", recData.value("Gender"));
        query.bindValue(":Height", recData.value("Height"));
        query.bindValue(":Birthday", recData.value("Birthday"));
    }
}

```

```

        query.bindValue(":Mobile", recData.value("Mobile"));
        query.bindValue(":Province", recData.value("Province"));
        query.bindValue(":City", recData.value("City"));
        query.bindValue(":Department", recData.value("Department"));
        query.bindValue(":Education", recData.value("Education"));
        query.bindValue(":Salary", recData.value("Salary"));
        query.bindValue(":Memo", recData.value("Memo"));
        query.bindValue(":Photo", recData.value("Photo"));
        if (!query.exec())
            QMessageBox::critical(this, "错误",
                                   "插入记录错误\n"+query.lastError().text(),
                                   QMessageBox::Ok, QMessageBox::NoButton);
        else //重新执行 SQL 语句查询
            qryModel->query().exec();
    }
    delete dataDialog;
}

```

程序首先用 QSqlQuery 类对象 query 执行一个 SQL 语句“select \* from employee where EmpNo =1”，这样不会查询到任何记录，目的就是得到一条空记录 curRec。

创建 WDialogData 类型的对话框 dataDialog 后，调用 setInsertRecord() 函数对话框进行初始化设置。

对话框 dataDialog 运行“确认”返回后，使用 query 执行 INSERT 语句插入一条新记录。若插入记录执行成功，需要重新执行数据模型 qryModel 的 SQL 语句查询数据，才会更新界面上的 tableView 的显示。

## 5. 删除记录

工具栏上的“删除记录”删除 tableView 上的当前记录，代码如下：

```

void MainWindow::on_actRecDelete_triggered()
{ //删除当前记录
    int curRecNo=theSelection->currentIndex().row();
    QSqlRecord curRec=qryModel->record(curRecNo); //获取当前记录
    if (curRec.isEmpty()) //当前为空记录
        return;

    int empNo=curRec.value("EmpNo").toInt(); //获取员工编号
    QSqlQuery query;
    query.prepare("delete from employee where EmpNo = :ID");
    query.bindValue(":ID", empNo);
    if (!query.exec())
        QMessageBox::critical(this, "错误",
                                "删除记录出现错误\n"+query.lastError().text(),
                                QMessageBox::Ok, QMessageBox::NoButton);
    else //重新执行 SQL 语句查询
        qryModel->query().exec();
}

```

从数据模型的当前记录中获取员工编号，然后使用一个 QSqlQuery 类的对象执行一条 DELETE 语句删除这条记录。

删除记录后需要重新设置数据模型 qryModel 的 SQL 语句并查询数据，以更新数据和 tableView 的显示。

## 6. 记录遍历

工具栏上的“涨工资”按钮通过记录遍历，修改所有记录的 Salary 字段的值，其实现代码如下：

```
void MainWindow::on_actScan_triggered()
{ //涨工资，记录遍历
    QSqlQuery qryEmpList;
    qryEmpList.exec("SELECT EmpNo,Salary FROM employee ORDER BY EmpNo");
    qryEmpList.first();
    QSqlQuery qryUpdate; //临时 QSqlQuery
    qryUpdate.prepare("UPDATE employee SET Salary=:Salary WHERE EmpNo = :ID");

    while (qryEmpList.isValid()) //当前记录有效
    {
        int empID=qryEmpList.value("EmpNo").toInt(); //获取 EmpNo
        float salary=qryEmpList.value("Salary").toFloat(); //获取 Salary
        salary=salary+1000; //涨工资
        qryUpdate.bindValue(":ID",empID);
        qryUpdate.bindValue(":Salary",salary); //设置 SQL 语句参数
        qryUpdate.exec(); //执行 update
        if (!qryEmpList.next()) //移动到下一条记录，并判断是否到末尾了
            break;
    }
    qryModel->query().exec(); //重新查询数据，更新 tableView 的显示
    QMessageBox::information(this, "提示", "涨工资计算完毕",
        QMessageBox::Ok, QMessageBox::NoButton);
}
```

程序里使用了两个 QSqlQuery 类变量，qryEmpList 查询 EmpNo 和 Salary 两个字段的全部记录，qryUpdate 用于执行一个带参数的 UPDATE 语句，每次更新一条记录。

遍历 qryEmpList 的所有记录，QSqlQuery 有 first()、previous()、next()、last()等函数记录移动，若到了最后一条记录后再执行 next()，将返回 false，以此判断是否遍历完所有记录。

# 11.5 QSqlRelationalTableModel 的使用

## 11.5.1 关系型数据表和实例功能

从图 11-2 的类继承关系图上看，QSqlRelationalTableModel 是 QSqlTableModel 的子类。QSqlRelationalTableModel 可以处理关系数据表，所谓关系数据表，是指将主表里的某个字段存储为代码型字段，而代码的具体意义在另外一个数据表里。

表 11-5 给出了 studInfo 的字段定义，studInfo 数据表实际存储的数据示例见表 11-12，字段 departID 和 majorID 存储的是学院代码和专业代码。

表 11-12 表 studInfo 的数据记录示例

序号	studID	name	gender	departID	majorID
1	2017200211	张三	男	20	2003
2	2017200102	李四	男	10	1002
3	2017300212	小明	男	50	5001
4	2017400102	小雅	女	50	5002

学院代码字段 departID 的具体意义在数据表 departments 里定义，departments 表的字段结构见表 11-3，departments 表的示例数据见表 11-13。

表 11-13 表 departments 的数据记录示例

序号	departID	department
1	10	生物科学学院
2	20	数理学院
3	50	化工学院

同样，专业数据表 majors 定义了专业代码及其专业名称，majors 表的字段结构见表 11-4，majors 表的示例数据见表 11-14。majors 数据表不仅定义了专业代码的意义，还使用了学院代码字段 departID，与 departments 数据表发生关联。

表 11-14 表 majors 的数据记录示例

序号	majorID	major	departID
1	1001	生物遗传学	10
2	1002	生物工程	10
3	5001	计算机网络	50
4	5002	自动化	50

在数据库设计中使用代码字段和代码表的好处：一是可以减少数据表的存储量，一个大的数据表存储代码远比存储具体文字用的存储空间少；二是代码表示的文字可能会被修改，例如学院的名称可能会修改，这时只需修改代码表里一条记录而已。

QSqlRelationalTableModel 类专门用来编辑这种具有代码字段的数据表，可以很方便地将代码字段与关系数据表建立关系，在显示和编辑数据表时，直接使用关系数据表的代码意义字段的内容。实例 samp11\_4 演示关系型数据表使用的方法，图 11-8 是程序运行界面。

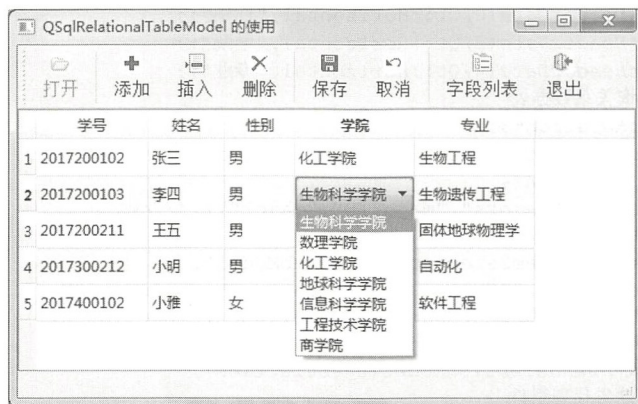


图 11-8 实例 samp11\_4 编辑数据表 studInfo 的界面

实例 samp11\_4 使用 QSqlRelationalTableModel 作为 tableView 的数据源，显示和编辑 studInfo 数据表。学院和专业两个字段与代码表建立了关系，tableView 中直接显示这两个字段的文字内容，编辑时有一个下拉列表框，列表框里是代码表里全部代码意义文字。



## 11.5.2 关系型数据模型功能实现

### 1. 主窗口类定义

主窗口类定义如下，定义了一个 QSqlRelationalTableModel 类型的变量 tabModel 作为数据模型。

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QSqlDatabase DB; //数据库连接
    QSqlRelationalTableModel *tabModel; //数据模型
    QItemSelectionModel *theSelection; //选择模型
    void openTable(); //打开数据表
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    void on_currentChanged(const QModelIndex &current, const QModelIndex &previous);
};
```

### 2. 打开数据表

主窗口工具栏的“打开”按钮选择数据库文件 demodb.db3，然后调用 openTable()打开数据表，openTable()函数的代码如下：

```
void MainWindow::openTable()
{ //打开数据表
    tabModel=new QSqlRelationalTableModel(this,DB);
    tabModel->setTable("studInfo"); //设置数据表
    tabModel->setEditStrategy(QSqlTableModel::OnManualSubmit);
    tabModel->setSort(0,Qt::AscendingOrder);
    tabModel->setHeaderData(0,Qt::Horizontal,"学号");
    tabModel->setHeaderData(1,Qt::Horizontal,"姓名");
    tabModel->setHeaderData(2,Qt::Horizontal,"性别");
    tabModel->setHeaderData(3,Qt::Horizontal,"学院");
    tabModel->setHeaderData(4,Qt::Horizontal,"专业");
    //设置代码字段的查询关系数据表
    tabModel->setRelation(3,
        QSqlRelation("departments","departID","department")); //学院
    tabModel->setRelation(4,
        QSqlRelation("majors","majorID","major")); //专业

    theSelection=new QItemSelectionModel(tabModel);
    connect(theSelection,SIGNAL(currentChanged(QModelIndex,QModelIndex)),
        this,SLOT(on_currentChanged(QModelIndex,QModelIndex)));
    ui->tableView->setModel(tabModel);
    ui->tableView->setSelectionModel(theSelection);
    //为关系型字段设置缺省代理组件
    ui->tableView->setItemDelegate(
        new QSqlRelationalDelegate(ui->tableView));
    tabModel->select(); //打开数据表
    ui->actOpenDB->setEnabled(false);
    ui->actRecAppend->setEnabled(true);
    ui->actRecInsert->setEnabled(true);
    ui->actRecDelete->setEnabled(true);
```

```

    ui->actFields->setEnabled(true);
}

```

QSqlRelationalTableModel 类的主要函数与 QSqlTableModel 相同，有一个新的函数 setRelation() 用于设置代码字段的关联数据表和关联字段。setRelation()函数的定义如下：

```
void setRelation(int column, const QSqlRelation &relation)
```

其中 column 是主表中代码字段的序号，relation 是 QSqlRelation 类型的表示关联数据表的关系。设置代码字段 departID 的关系的代码如下：

```

tabModel->setRelation(3,
    QSqlRelation("departments", "departID", "department"));

```

第 1 个参数（数字 3）是 departID 字段在 studInfo 表中的字段序号。

第 2 个参数用 QSqlRelation("departments", "departID", "department") 创建了一个 QSqlRelation 类型对象，其中，参数 "departments" 是代码表 departments，参数 "departID" 是代码字段名称，参数 "department" 是代码意义字段名称。

openTable() 函数中还有关键的一行：

```
ui->tableView->setItemDelegate(new QSqlRelationalDelegate(ui->tableView));
```

这是在 tableView 中为代码字段创建缺省的关系型代理组件，这样在 tableView 中编辑代码字段的内容时，才会出现一个下拉列表框，列出代码表的所有可选内容。

### 3. 实际字段列表

使用 QSqlRelationalTableModel 类设置代码字段的关系后，在 tableView 中以代码意义显示代码字段的内容，其实际字段有没有变化呢？

工具栏上的按钮“字段列表”列出了 tabModel 的所有字段的名称，其代码如下：

```

void MainWindow::on_actFields_triggered()
{ // 获取字段列表
    QSqlRecord emptyRec=tabModel->record(); // 获取空记录，只有字段名
    QString str;
    for (int i=0; i<emptyRec.count(); i++)
        str=str+emptyRec.fieldName(i)+'\n';
    QMessageBox::information(this, "所有字段名", str,
        QMessageBox::Ok, QMessageBox::NoButton);
}

```

字段列表的对话框如图 11-9 所示，可以看到两个代码字段 departID 和 majorID 被代码表中的代码意义字段 department 和 major 替换了。但是在界面上修改数据后，数据能以代码的形式保存到原数据表里。

### 4. 其他功能的实现

工具栏上其他“添加”“插入”“删除”“保存”“取消”等按钮的功能的实现与实例 samp11\_1 相同，自定义槽函数 on\_currentChanged() 的实现也与 samp11\_1 相同，这些功能的实现就不具体介绍了。

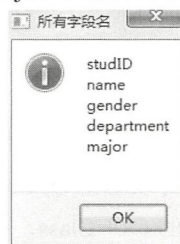


图 11-9 tabModel 的字段列表

## 第 12 章

# 自定义插件和库

当 UI 设计器提供的界面组件不满足实际设计需求时，可以从 `QWidget` 继承自定义界面组件。有两种方法使用自定义界面组件，一种是提升法 (promotion)，例如在 8.3 节将一个 `QGraphicsView` 组件提升为自定义的 `QWGraphicsView` 类，提升法用于界面可视化设计时不够直观，不能在界面上即刻显示自定义组件的效果；另一种是为 UI 设计器设计自定义界面组件的 Widget 插件，直接安装到 UI 设计器的组件面板里，如同 Qt 自带的界面设计组件一样使用，在设计时就能看到组件的实际显示效果，只是编译和运行时需要使用到插件的动态链接库 (Windows 平台上)。

本章先介绍这两种自定义 Widget 组件的设计和使用方法，再介绍 Qt 编写和使用静态链接库和共享库 (Windows 平台上就是动态链接库) 的方法。

## 12.1 自定义 Widget 组件

### 12.1.1 自定义 Widget 子类 `QmyBattery`

Qt 的 UI 设计器提供了很多 GUI 设计的界面组件，可以满足常见的界面设计需求。但是某些时候需要设计特殊的界面组件，而在 UI 设计器的组件面板里根本没有合适的组件，这时就需要设计自定义的界面组件。

所有界面组件的基类是 `QWidget`，要设计自定义的界面组件，可以从 `QWidget` 继承一个自定义的类，重定义其 `paintEvent()` 事件，利用 Qt 的绘图功能绘制组件外观，并实现需要的其他功能。

例如，假设需要设计一个如图 12-1 所示的电池电量显示组件，用于电池使用或充电时显示其电量，但是在 UI 设计器的组件面板里是没有这样一个现成的组件的。这就需要设计一个自定义的 Widget 组件。

为此，设计一个从 `QWidget` 继承的类 `QmyBattery`。创建 C++ 类，可以单击 Qt Creator 的 “File” → “New File or Project” 菜单项，在出现的对话框里选择 C++ 组里的 C++ Class，在向导中设置类的名称，并选择基类为 `QWidget`。

定义 `QmyBattery` 类的 `qmybattery.h` 文件的完整代码如下：

```
#include <QWidget>
#include <QColor>
```

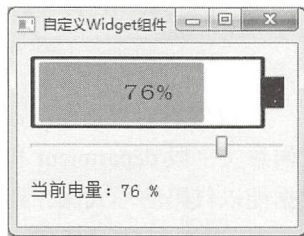


图 12-1 实例 samp12\_1 运行时界面

```

class QmyBattery : public QWidget
{
    Q_OBJECT
private:
    QColor    mColorBack=Qt::white;    //背景颜色
    QColor    mColorBorder=Qt::black;  //电池边框颜色
    QColor    mColorPower=Qt::green;   //电量柱颜色
    QColor    mColorWarning=Qt::red;   //电量短缺时的颜色
    int       mPowerLevel=60;           //电量 0-100
    int       mWarnLevel=20;            //电量低警示阈值
protected:
    void      paintEvent(QPaintEvent *event) Q_DECL_OVERRIDE;
public:
    explicit QmyBattery(QWidget *parent = 0);
    void      setPowerLevel(int pow);    //设置当前电量
    int       powerLevel();
    void      setWarnLevel(int warn);    //设置电量低阈值
    int       warnLevel();
    QSize     sizeHint();                //缺省大小
signals:
    void      powerLevelChanged(int );
public slots:
};

```

在 `private` 部分定义了几个私有变量，主要是各种颜色的定义、当前电量值 `mPowerLevel` 和电量低阈值 `mWarnLevel`。

`protected` 部分重定义了 `paintEvent()` 事件，在第 8 章中介绍过，`QWidget` 类的 `paintEvent()` 事件用于界面绘制，在此事件里，可以使用 `QPainter` 的各种绘图功能绘制自己需要的界面。

`public` 部分定义了用于读取和设置当前电量值、电量低阈值的函数，还定义了 `sizeHint()` 函数，用于返回组件缺省大小。

定义了一个信号 `powerLevelChanged(int)`，在当前电量值改变时发射该信号，使用 `QmyBattery` 类时可以设计槽函数对此信号做处理。

下面是 `QmyBattery` 类的实现代码，复杂一点的部分是 `paintEvent()` 事件函数里绘制界面的功能实现，这里设置了窗口逻辑坐标，所以，当组件大小变化时，绘制的电池大小也会自动变化。`QPainter` 绘图的功能在第 8 章有详细介绍，这里不再详细解释。

```

#include "qmybattery.h"
#include <QPainter>
void QmyBattery::paintEvent(QPaintEvent *event)
{
    //界面组件的绘制
    Q_UNUSED(event);
    QPainter painter(this);
    QRect rect(0,0,width(),height());
    painter.setViewport(rect); //设置 Viewport
    painter.setWindow(0,0,120,50); // 设置窗口大小，逻辑坐标
    painter.setRenderHint(QPainter::Antialiasing);
    painter.setRenderHint(QPainter::TextAntialiasing);
    //绘制电池边框
    QPen pen; //设置画笔
    pen.setWidth(2);
    pen.setColor(mColorBorder);
}

```



```

    pen.setStyle(Qt::SolidLine);
    pen.setCapStyle(Qt::FlatCap);
    pen.setJoinStyle(Qt::BevelJoin);
    painter.setPen(pen);
    QBrush brush;//画刷
    brush.setColor(mColorBack);
    brush.setStyle(Qt::SolidPattern);
    painter.setBrush(brush);
    rect.setRect(1,1,109,48);
    painter.drawRect(rect);//绘制电池边框
    brush.setColor(mColorBorder);
    painter.setBrush(brush);
    rect.setRect(110,15,10,20);
    painter.drawRect(rect); //画电池正极头
//画电池柱
    if (mPowerLevel>mWarnLevel)
    { //正常颜色电量柱
        brush.setColor(mColorPower);
        pen.setColor(mColorPower);
    }
    else
    { //电量低电量柱
        brush.setColor(mColorWarning);
        pen.setColor(mColorWarning);
    }
    painter.setBrush(brush);
    painter.setPen(pen);
    if (mPowerLevel>0)
    {
        rect.setRect(5,5,mPowerLevel,40);
        painter.drawRect(rect);//画电池柱
    }
//绘制电量百分比文字
    QFontMetrics textSize(this->font());
    QString powStr=QString::asprintf("%d%%",mPowerLevel);
    QRect textRect=textSize.boundingRect(powStr);//得到字符串的 rect
    painter.setFont(this->font());
    pen.setColor(mColorBorder);
    painter.setPen(pen);
    painter.drawText(55-textRect.width()/2,
        23+textRect.height()/2, powStr);
}

void QmyBattery::setPowerLevel(int pow)
{ //设置当前电量值
    mPowerLevel=pow;
    emit powerLevelChanged(pow); //发射信号
    repaint();
}

int QmyBattery::powerLevel()
{ //返回当前电量值
    return mPowerLevel;
}

void QmyBattery::setWarnLevel(int warn)

```

```

{ //设置电量低阈值
    mWarnLevel=warn;
    repaint();
}

int QmyBattery::warnLevel()
{ //返回电量低阈值
    return mWarnLevel;
}

QSize QmyBattery::sizeHint()
{ //缺省大小, 调整比例
    int H=this->height();
    int W=H*12/5;
    QSize size(W,H);
    return size;
}

```

## 12.1.2 自定义 Widget 组件的使用

实现了 QmyBattery 类之后, 若是用代码创建 QmyBattery 类对象, 其使用与一般的组件类是一样的; 若是在 UI 设计器中使用 QmyBattery, 则需要采用提升法 (promotion)。

实例 samp12\_1 是一个基于 QWidget 的应用程序, 使用 UI 设计器设计主窗体时, 在窗体上放置一个 QWidget 类组件, 然后鼠标右键调出其快捷菜单, 单击 “Promote to” 菜单项, 会出现如图 12-2 所示的对话框。

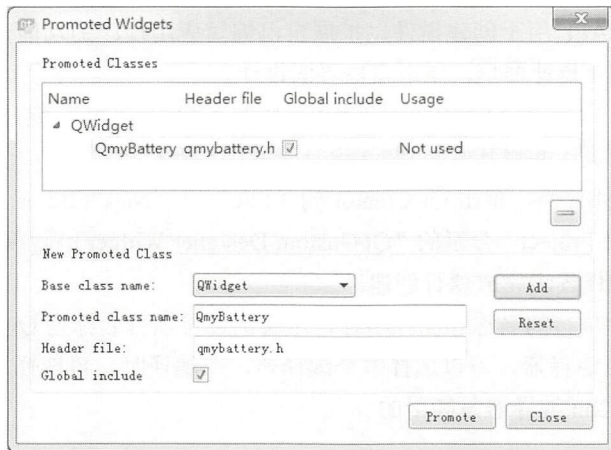


图 12-2 Widget 组件提升对话框

此对话框里, 在基类名称下拉列表框里选择 QWidget, 将提升后的类名称设置为 QmyBattery, 头文件名称会自动生成。可以将设置添加到已提升类的列表里, 以便重复使用。设置后, 单击 “Promote” 按钮, 就可以将此 QWidget 组件提升为 QmyBattery 类。提升后, 在 Property Editor 里, 会看到这个组件的类名称变为了 QmyBattery。然后, 将其 objectName 更改为 battery。

虽然界面上放置的 QWidget 组件被提升为了 QmyBattery 类, 但是在这个组件的 “Go to slot” 对话框里并没有 QmyBattery 类的 powerLevelChanged(int) 信号, 无法采用可视化方法生成信号的

槽函数。

在主窗口上放置一个 QSlider 组件和一个 QLabel 组件。滑动标尺改变数值时，设置为 battery 的当前电量值，其 valueChanged()信号的槽函数代码如下：

```
void MainWindow::on_slider_valueChanged(int value)
{ //滑动改变电量值
    ui->battery->setPowerLevel(value);
    QString str=QStringLiteral("当前电量:")+QString::asprintf("%d %%",value);
    ui->LabInfo->setText(str);
}
```

实例运行时就可以得到如图 12-1 所示的运行界面。battery 的各种参数采用其缺省的设置，battery 的当前电量值改变时，内部会调用 paintEvent()事件代码重新绘制电池显示效果。

## 12.2 自定义 Qt Designer 插件

### 12.2.1 创建 Qt Designer Widget 插件项目

Qt 提供两种设计插件的 API，可以用于扩展 Qt 的功能。高级（high-level）API 用于设计插件以扩展 Qt 的功能，例如定制数据库驱动、图像格式、文本编码、定制样式等，Qt Creator 里大量采用了插件，单击 Qt Creator 的主菜单栏的“Help”→“About Plugins”菜单项，会显示 Qt Creator 里已经安装的各种插件。

低级（low-level）API 用于创建插件以扩展自己编写应用程序的功能，最常见的就是将自定义 Widget 组件安装到 UI 设计器里，用于窗口界面设计。

本节创建一个与 12.1 节的 QmyBattery 功能一样的类 QwBattery，但是采用创建 Qt Designer 插件的方式来创建这个类，并将其安装到 UI 设计器的组件面板里。

要创建 UI 设计器插件类，单击 Qt Creator 的“File”→“New File or Project”菜单，在出现的对话框里选择“Other Project”分组的“Qt Custom Designer Widget”项目，会出现一个向导对话框。按照这个向导的操作逐步完成项目创建。

第 1 步是设置插件项目的名称和保存路径，本实例设置项目名称为 QwBatteryPlugin。

第 2 步是选择项目编译器，可以选择多个编译器，在编译时，再选择具体的编译器。但是实际上只有 MSVC2015 32bit 编译器是能用的。

---

**注意** 使用 Qt 创建的 Widget 插件，若要在 Qt Creator 的 UI 设计器里正常显示，编译插件的编译器版本必须和编译 Qt Creator 的版本一致。

---

Qt 5.9 的 Qt Creator 是基于 MSVC2015 32bit 编译器编译的（单击 Qt Creator 的“Help”→“About Qt Creator”菜单，出现的对话框里会显示 Qt Creator 的版本信息和使用的编译器信息）。所以，为了在 Qt Creator 里设计窗体时能够正常显示插件，只能使用 Qt 5.9 MSVC2015 32bit 编译器。

第 3 步是设置自定义 QWidget 类的名称（见图 12-3），只需在左侧的 Widget classes 列表里设置类名，右侧就会自动设置缺省的文件名，这里添加一个类 QwBattery。还可以选择一个图标文件

作为自定义组件在 UI 设计器组件面板里的显示图标。

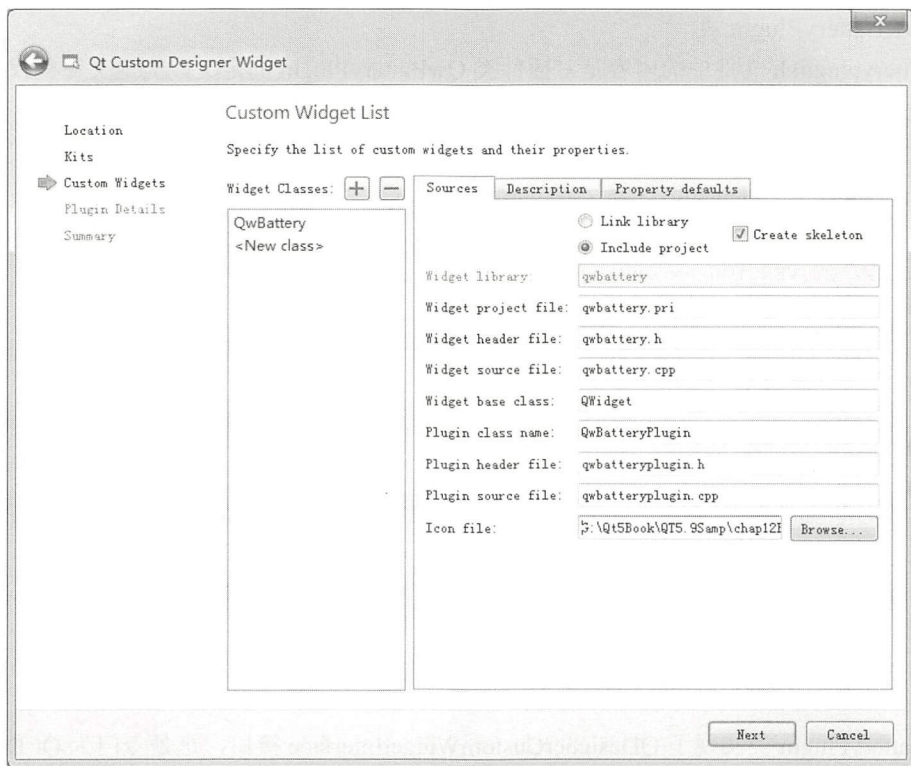


图 12-3 设置界面组件类的名称

在图 12-3 的 Description 页还可以设置 Group、Tooltip 和 What's this 等信息，Group 是自定义组件在组件面板里的分组名称，这里设置为“My Widget”。

第 4 步是显示和设置插件、资源文件名称。本实例缺省的插件名称是 `qwbatteryplugin`，资源文件名称为 `icons.qrc`，一般用缺省的即可。

第 5 步，完成设置，生成项目。

完成设置后生成的项目的文件组织结构如图 12-4 所示，这些文件包括以下几个。

- `QwBatteryPlugin.pro` 是插件项目的项目文件，用于实现插件接口。
- `qwbatteryplugin.h` 和 `qwbatteryplugin.cpp` 是插件的头文件和实现文件。
- `icons.qrc` 是插件项目的资源文件，存储了图标。
- `qwbattery.pri` 是包含在 `QwBatteryPlugin.pro` 项目中的一个项目文件（图 12-3 中选择“Include project”），用于管理自定义组件类。
- `qwbattery.h` 和 `qwbattery.cpp` 是自定义类 `QwBattery` 的头文件和实现文件。

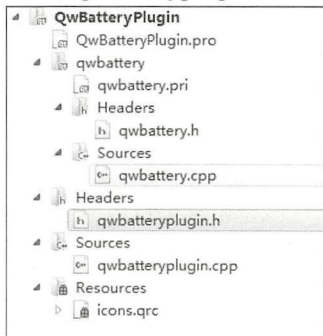


图 12-4 插件项目文件组织结构



## 12.2.2 插件项目各文件的功能实现

### 1. QwBatteryPlugin 类

qwbatteryplugin.h 文件中的内容是对插件类 QwBatteryPlugin 的定义，类定义完整代码如下：

```
#include <QDesignerCustomWidgetInterface>
class QwBatteryPlugin : public QObject, public QDesignerCustomWidgetInterface
{
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetInterface)
#ifdef QT_VERSION >= 0x050000
    Q_PLUGIN_METADATA(IID "org.qt-project.Qt.QDesignerCustomWidgetInterface")
#endif // QT_VERSION >= 0x050000
public:
    QwBatteryPlugin(QObject *parent = 0);
    bool isContainer() const;
    bool isInitialized() const;
    QIcon icon() const;
    QString domXml() const;
    QString group() const;
    QString includeFile() const;
    QString name() const;
    QString tooltip() const;
    QString whatsThis() const;
    QWidget *createWidget(QWidget *parent);
    void initialize(QDesignerFormEditorInterface *core);
private:
    bool m_initialized;
};
```

QwBatteryPlugin 类实现了 QDesignerCustomWidgetInterface 接口，这是专门为 Qt Designer 设计自定义 Widget 组件的接口。

在这个类定义里，除了 Q\_OBJECT 宏之外，还用了 Q\_INTERFACES 宏声明了实现的接口，用 Q\_PLUGIN\_METADATA 声明了元数据名称，这些都无需改动。

public 部分的函数都是有关插件信息或功能的一些函数，通过其实现代码可以看出这些函数的功能。下面是 qwbatteryplugin.cpp 文件里的实现代码。

```
#include "qwbattery.h"
#include "qwbatteryplugin.h"
#include <QtPlugin>
QwBatteryPlugin::QwBatteryPlugin(QObject *parent) : QObject(parent)
{
    m_initialized = false;
}

void QwBatteryPlugin::initialize(QDesignerFormEditorInterface * /* core */)
{
    if (m_initialized)
        return;
    // Add extension registrations, etc. here
    m_initialized = true;
}

bool QwBatteryPlugin::isInitialized() const
{
    //是否初始化
    return m_initialized;
}

QWidget *QwBatteryPlugin::createWidget(QWidget *parent)
```

```

    //返回自定义 widget 组件的实例
    return new QwBattery(parent);
}
QString QwBatteryPlugin::name() const
//自定义 widget 组件类的名称
    return QLatin1String("QwBattery");
}
QString QwBatteryPlugin::group() const
//在组件面板中所属分组名称
    return QLatin1String("MyWidget");
}
QIcon QwBatteryPlugin::icon() const
//图标文件名
    return QIcon(QLatin1String(":/44.ico"));
}
QString QwBatteryPlugin::toolTip() const
//toolTip 信息
    return QLatin1String("Battery charger indicator");
}
QString QwBatteryPlugin::whatsThis() const
//whatsThis 信息
    return QLatin1String("A battery charger indicator");
}
bool QwBatteryPlugin::isContainer() const
{ //是否作为容器, false 表示该组件上不允许再放其他组件
    return false;
}
QString QwBatteryPlugin::domXml() const
//XML 文件描述信息
    return QLatin1String("<widget class=\"QwBattery\" name=\"qwBattery\">\n</widget>\n");
}
QString QwBatteryPlugin::includeFile() const
//包含文件名
    return QLatin1String("qwbattery.h");
}
#ifdef QT_VERSION < 0x050000
Q_EXPORT_PLUGIN2(qwbatteryplugin, QwBatteryPlugin)
#endif // QT_VERSION < 0x050000

```

这些函数的部分内容是根据创建插件向导里设置的内容自动生成的。createWidget()函数创建一个 QwBattery 类的实例, 在 UI 设计器里作为设计实例; name()函数返回组件的类名称; group()函数设置组件安装在面板里的分组名称; icon()设置组件的图标; isContainer()设置组件是否作为容器, false 表示不作为容器, 不能在这个组件上放置其他组件; domXml()函数用 XML 设置组件的一些属性, 缺省的只设置了类名和实例名。

## 2. QwBatteryPlugin.pro 的内容

QwBatteryPlugin.pro 是插件项目的项目管理文件, 其内容如下:

```

CONFIG       += plugin debug_and_release
TARGET       = $$qtLibraryTarget(qwbatteryplugin)
TEMPLATE     = lib

HEADERS      = qwbatteryplugin.h
SOURCES      = qwbatteryplugin.cpp

```

```

RESOURCES    = icons.qrc
LIBS         += -L.

greaterThan(Qt_MAJOR_VERSION, 4) {
    Qt += designer
} else {
    CONFIG += designer
}

target.path = $$[Qt_INSTALL_PLUGINS]/designer
INSTALLS    += target
include(qwbattery.pri)

```

CONFIG 是用于 qkame 编译设置的，这里配置为：

```
CONFIG += plugin debug_and_release
```

其中，plugin 表示项目要作为插件，编译后只会产生 lib 和 dll (或.so) 文件，debug\_and\_release 表示项目可以用 debug 和 release 模式编译。

TEMPLATE 定义项目的类型，这里设置为：

```
TEMPLATE = lib
```

这表示项目是一个库，一般的应用程序模板类型是 app。

### 3. 内置项目 qwbattery.pri

qwbattery.pri 是内置于 QwBatteryPlugin.pro 中的项目，qwbattery.pri 项目配置文件只有两行，也就是这个内置项目中包含的头文件和源文件名称。

```

HEADERS += qwbattery.h
SOURCES += qwbattery.cpp

```

### 4. 组件类 QwBattery 的定义

qwbattery.h 里的内容是对组件类 QwBattery 的类定义，其功能与 12.1 节中的 QmyBattery 类完全一样。这两个类的名称之所以不同，是为了在编译两个实例时不产生冲突。

QwBattery 类的定义与 QmyBattery 的定义基本一样，只是在声明类的时候需要加一个宏 QDESIGNER\_WIDGET\_EXPORT，并且用 Q\_PROPERTY 宏定义了一个属性 powerLevel。QwBattery 类的完整定义如下：

```

#include <QDesignerExportWidget>
#include <QWidget>
class QDESIGNER_WIDGET_EXPORT QwBattery : public QWidget
{
    Q_OBJECT
    //自定义属性
    Q_PROPERTY(int powerLevel READ powerLevel WRITE setPowerLevel
NOTIFY powerLevelChanged DESIGNABLE true)
private:
    QColor mColorBack=Qt::white;    //背景颜色
    QColor mColorBorder=Qt::black;  //电池边框颜色
    QColor mColorPower=Qt::green;   //电量柱颜色
    QColor mColorWarning=Qt::red;   //电量短缺时的颜色
    int mPowerLevel=60;              //电量 0-100
    int mWarnLevel=20;               //电量低警示阈值
protected:

```

```

void    paintEvent(QPaintEvent *event) Q_DECL_OVERRIDE;
public:
    explicit QwBattery(QWidget *parent = 0);
    void    setPowerLevel(int pow); //设置当前电量
    int     powerLevel();
    void    setWarnLevel(int warn); //设置电量低阈值
    int     warnLevel();
    QSize   sizeHint(); //报告缺省大小
signals:
    void    powerLevelChanged(int );
public slots:
};

```

QDESIGNER\_WIDGET\_EXPORT 宏用于将自定义组件类从插件导出给 Qt Designer 使用，必须在类名称前使用此宏。

Q\_PROPERTY 宏用于定义属性，这里定义了一个 int 类型的属性 powerLevel。READ 宏声明了属性的读取函数是 powerLevel(); WRITE 宏声明了设置属性值的函数是 setPowerLevel(); NOTIFY 宏声明了其值变化时发射的信号是 powerLevelChanged(); DESIGNABLE 宏定义属性在 UI 设计器里是否可见，缺省为 true。

将从 QWidget 继承的子类 QwBattery 作为插件安装到 UI 设计器的组件面板里，则在设计期间就可以从属性编辑器里看到这个 powerLevel 属性并进行设置。

QwBattery 类的实现代码与 QmyBattery 的实现代码完全相同，不再列出。

## 12.2.3 插件的编译与安装

使用 MSVC2015 32bit 编译器，将插件项目在 release 模式下编译，编译后会生成 qwbatteryplugin.dll 和 qwbatteryplugin.lib 两个文件。

qwbatteryplugin.dll 是插件的动态链接库文件，需要将此文件复制到 Qt Creator 的插件目录和 Qt 的插件目录下。例如，要把 Qt 安装到 D:\Qt\Qt5.9.1 目录下，就需要将 qwbatteryplugin.dll 复制到如下两个目录下：

```

D:\Qt\Qt5.9.1\Tools\QtCreator\bin\plugins\designer
D:\Qt\Qt5.9.1\5.9.1\msvc2015\plugins\designer

```

重启 Qt Creator，使用 UI 设计器设计窗口时，在左侧的组件面板里会看到增加了一个“MyWidget”分组，里面有一个组件 QwBattery。

编译和安装 Widget 插件必须注意以下事项。

- 要让插件在 Qt creator 的 UI 设计器里正常显示，编译插件项目的编译器必须与编译 Qt Creator 的编译器一致，否则，即使将编译后生成的 DLL 文件复制到 Qt 的目录下，Qt Creator 的 UI 设计器的组件面板里也不会出现自定义的组件。例如，Qt Creator 4.3.1 是基于 Qt 5.9.1 和 MSVC2015 32 bit 编译器（单击 Qt Creator 的“Help”→“About Qt Creator”菜单项可以看到这些信息），那么编译插件就必须使用 Qt 5.9.1 MSVC2015 32 bit 编译器。
- 用 debug 和 release 模式编译的插件也分别只适用于 debug 和 release 模式编译的应用程序。在 debug 模式下编译的插件项目生成的 Lib 和 DLL 文件会在文件名最后自动增加一个字母



“d”，例如，本项目在 debug 模式下编译生成的是 qwbatteryplugind.dll 和 qwbatteryplugind.lib，这两个文件应用于使用此插件的应用程序的 debug 模式。

### 12.2.4 使用自定义插件

在 Qt Creator 的 UI 设计器的组件面板里能正常显示自定义的 QwBattery 组件后，就可以在窗体设计时使用 QwBattery 组件了。

创建一个基于 QWidget 的实例应用程序 BatteryUser。设计窗体时，从组件面板上拖放一个 QwBattery 到窗体上，窗体的功能与实例 samp12\_1 的窗体相同，但是在设计窗体时，就能直接看到 QwBattery 绘制的电池图形，在属性编辑器里可以编辑 QwBattery 组件的 powerLevel 属性（见图 12-5），在其“Go to slot”对话框里会出现自定义的信号 powerLevelChanged(int)，可以为此信号设计槽函数。

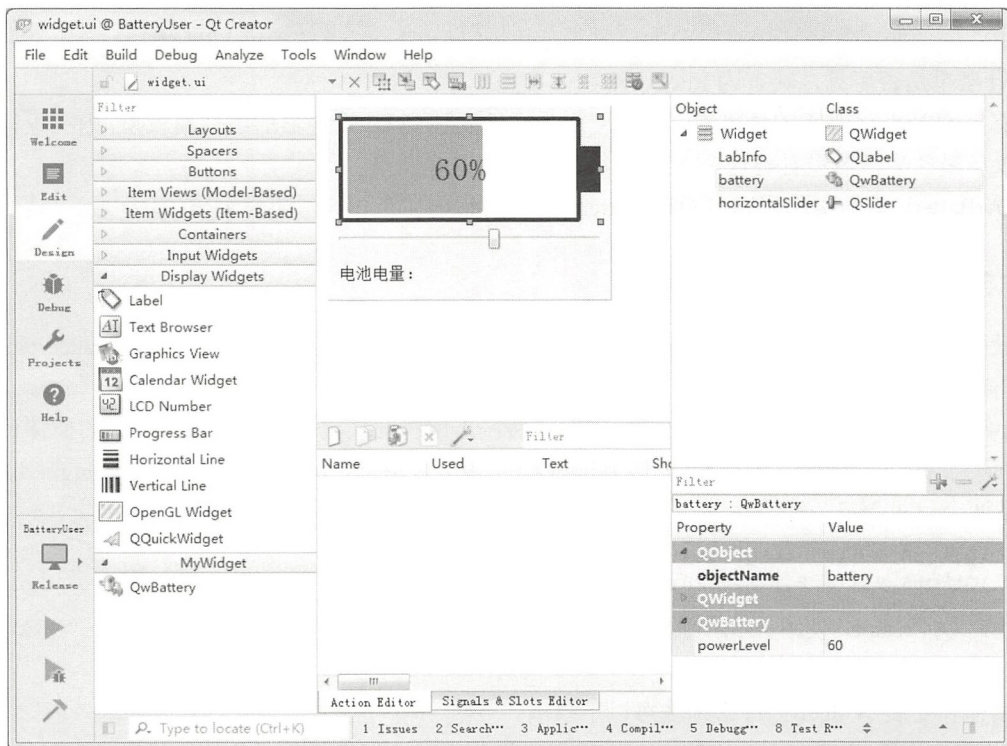


图 12-5 使用 QwBattery 组件的窗体设计

下面的代码实现的是利用滑动条设置 battery 的当前电量值，在 battery 的 powerLevelChanged() 信号的槽函数里，将当前电量值显示在标签里，程序运行后就可以实现与图 12-1 相同的功能。

```
void MainWindow::on_horizontalSlider_valueChanged(int value)
{ //拖动 slider 改变 battery 的电量值
    ui->battery->setPowerLevel(value);
}
```

```
void MainWindow::on_battery_powerLevelChanged(int arg1)
{ //电量值改变时，在标签中显示
    QString str=QStringLiteral("当前电量:")+QString::asprintf("%d %%",arg1);
    ui->LabInfo->setText(str);
}
```

注意 项目 BatteryUser 只能用 MSVC2015 32bit 编译器进行编译，因为使用的 Widget 插件类 QwBattery 是用 MSVC2015 32bit 编译的。

要正常编译项目 BatteryUser，还需要做以下设置。

- 在项目的源文件目录下创建一个 include 子目录（名称随个人喜好设置），将 QwBattery 类定义的头文件 qwbattery.h、插件的 debug 和 release 两种模式编译生成的库文件 qwbatteryplugin.lib 以及 qwbatteryplugind.lib 复制到此目录下，项目在编译链接时需要使用此头文件和库文件。
- 在项目管理器中，选中 BatteryUser 项目节点并单击右键，在快捷菜单中单击“Add Library...”，在出现的向导对话框第一步中，选择库类型时，将外部库“External Library”选中，因为本项目需要使用的是已经编译好的库文件。
- 在向导的第二步（见图 12-6），单击“Library file”编辑框后面的按钮，选择 include 目录下的库文件 qwbatteryplugin.lib，会自动填充“Include path”编辑框。在平台选择中可以只选择一个 windows 平台，连接方式选择 Dynamic，下方的 Add “d” suffix for debug version 表示在 debug 版本的库名称后面添加一个字母“d”，以便编译器自动区分 release 和 debug 版本的库文件。

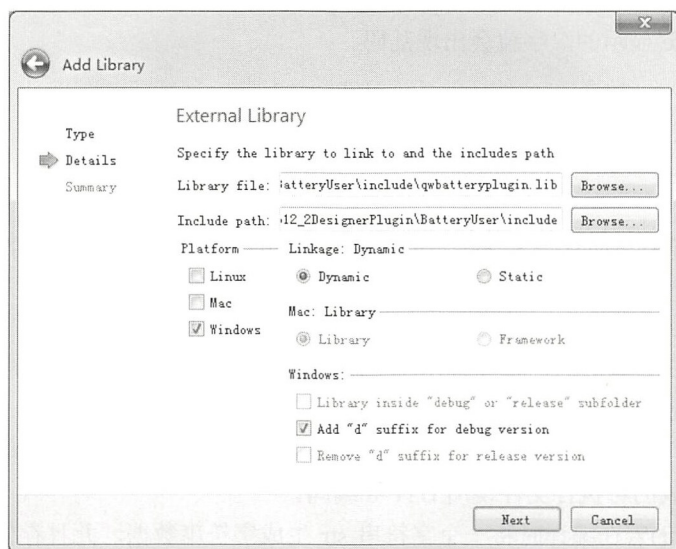


图 12-6 向项目添加插件的库文件

完成“Add Library”对话框的设置后，Qt Creator 会自动修改项目文件 BatteryUser.pro 的内容，在其中添加了以下几行：

```
win32:CONFIG(release, debug|release): LIBS += -L$${PWD}/include/ -lqwbatteryplugin
```

```
else:win32:CONFIG(debug, debug|release): LIBS += -L$$PWD/include/ -lqwbatterypluginind
INCLUDEPATH += $$PWD/include
DEPENDPATH += $$PWD/include
```

LIBS 用于设置添加的库文件，会判断当前项目是以 debug 还是 release 模式编译，自动加入 qwbatteryplugin.lib 或 qwbatterypluginind.lib 库文件。

INCLUDEPATH 和 DEPENDPATH 用于设置头文件目录和项目依赖项目目录，都指向项目路径下的 include 目录。

这样设置后，项目就可以在 release 或 debug 模式下编译了，同样只能使用 MSVC2015 32bit 编译器。

---

注意 要运行应用程序，还需要将插件的 DLL 文件复制到编译后的 release 或 debug 版本的可执行文件目录下，在本例中就是 qwbatteryplugin.dll 文件或 qwbatterypluginind.dll 文件，因为应用程序运行需要相应的 DLL 文件。在应用程序发布时，也需要将 DLL 文件随同应用程序发布。

---

自定义 Widget 插件的功能使得我们可以扩展 Qt Creator 的组件种类，设计自己需要的组件。也有许多第三方 Widget 插件可供直接使用，减少自己编程的工作量，例如 QWT 就是一套非常好的开源 Widget 插件。

## 12.2.5 使用 MSVC 编译器输出中文的问题

在 Qt Creator 中使用 MSVC 编译器编译项目时，若处理不当容易出现中文字符串乱码问题。例如 BatteryUser 项目中，如果槽函数 on\_battery\_powerLevelChanged() 的代码改写为如下的形式，程序运行时，LabInfo 显示的汉字就会出现乱码。

```
void MainWindow::on_battery_powerLevelChanged(int arg1)
{ //电量值改变时，在标签中显示
    QString str="当前电量: "+QString::asprintf("%d %%",arg1);
    ui->LabInfo->setText(str);
}
```

这是因为 Qt Creator 保存的文件使用的是 UTF-8 编码（是任何平台、任何语言都可以使用的跨平台的字符集），MSVC 编译器虽然可以正常编译带 BOM 的 UTF-8 编码的源文件，但是生成的可执行文件的编码是 Windows 本地字符集，比如 GB2312。也就是在可执行文件中，字符串“当前电量:”是以 GB2312 编码的，而可执行程序执行到这条语句时，对这个字符串却是以 UTF-8 解码的，这样就会出现乱码。

解决这个问题有两种方法，一种方法是使用 QStringLiteral() 宏封装字符串，另一种方法是强制 MSVC 编译器生成的可执行文件使用 UTF-8 编码。

QStringLiteral(str) 宏在编译时将一个字符串 str 生成字符串数据，并且存储在编译后文件的只读数据段中，程序运行时使用到此字符串时，只需读出此字符串数据即可。所以，BatteryUser 项目中槽函数 on\_battery\_powerLevelChanged() 中使用的一行代码如下：

```
QString str=QStringLiteral("当前电量:")+QString::asprintf("%d %%",arg1);
```

程序中需要使用 QStringLiteral() 宏对每个中文字符串进行封装，并且不能再使用 tr() 函数用于



翻译字符串。

强制 MSVC 编译器采用 UTF-8 编码生成可执行文件，需要在每个使用到中文字符串的头文件和源程序文件的前部加入如下的语句：

```
#if _MSC_VER >= 1600    //MSVC2015>1899,    MSVC_VER=14.0
#pragma execution_character_set("utf-8")
#endif
```

MSVC2010 以后的编译器可以使用此方案，这是强制编译后的执行文件采用 UTF-8 编码。这样，即使不再使用 QStringLiteral()宏，程序运行时也不会再出现汉字乱码的问题了。而且，也可以采用 tr()函数用于翻译字符串。

## 12.3 创建和使用静态链接库

### 12.3.1 创建静态链接库

创建一个静态链接库项目，设计各种需要导出的类，包括具有 UI 的窗体类、对话框类，编译后可以生成一个 lib 文件（MSVC 编译器生成后缀为“.lib”的文件，MinGW 编译器生成后缀为“.a”的文件），在另一个应用程序里使用这个 lib 文件和类的头文件（不需要 cpp 源文件），就可以静态编译到应用程序里。这种方式适合于在小组开发时，每个人负责自己的部分，使用其他人设计的代码时只能使用而不能看到或修改源代码，便于项目代码的管理。

创建静态链接库项目，单击 Qt Creator 的“File”→“New File or Project”菜单项，在出现的“New File or Project”对话框中选择 Projects 组里的 Library，在右侧的具体类别中再选择 C++ Library，单击“Choose...”按钮后出现如图 12-7 所示的向导对话框。

在此对话框的 Type 下拉列表框里选择“Statically Linked Library”，并给项目命名，例如 myStaticLib，再选择项目保存目录。单击“Next”按钮后选择编译器，再下一步选择需要包含的 Qt 模块，再下一步是类定义页面（见图 12-8），在其中输入类的名称。

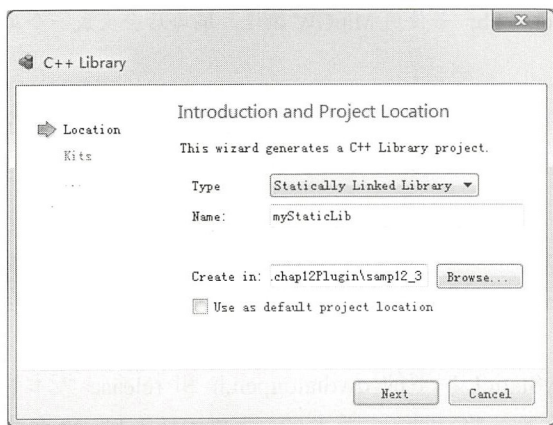


图 12-7 创建静态链接库项目向导对话框

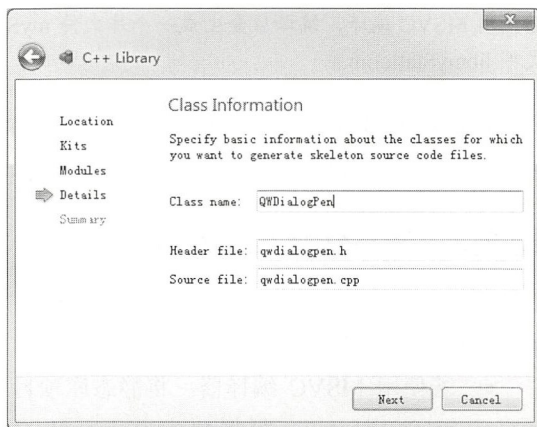


图 12-8 创建静态链接库项目向导的类信息页面



本实例将 9.3 节设计的一个 QPen 属性设置对话框 QWDialogPen 作为静态库的导出类，所以在图 12-8 的类定义界面上输入类名称为 QWDialogPen，头文件和源程序文件名会自动生成。在图 12-8 中单击“Next”按钮，再下一步结束即可。

这样生成的静态库项目 myStaticLib 包括 3 个文件：myStaticLib.pro、qwdialogpen.h 和 qwdialogpen.cpp。

我们希望将 9.3 节设计的一个 QPen 属性设置对话框 QWDialogPen 作为静态库的类，为此将 9.3 节 QWDialogPen 类相关的 3 个文件 qwdialogpen.h、qwdialogpen.cpp 和 qwdialogpen.ui 复制到 myStaticLib 项目的源文件目录下，覆盖自动生成的两个文件，并且将 qwdialogpen.ui 添加到项目中。

QWDialogPen 类相关的 3 个文件在 9.3 节有详细介绍，添加到静态库项目 myStaticLib 之后无需做任何修改，所以其内容不再详述。

项目配置文件 myStaticLib.pro 是对本项目的设置，其内容如下：

```
QT      += widgets
TARGET  = myStaticLib
TEMPLATE = lib
CONFIG += staticlib

SOURCES += qwdialogpen.cpp
HEADERS += qwdialogpen.h
unix {
    target.path = /usr/lib
    INSTALLS += target
}
FORMS += qwdialogpen.ui
```

TEMPLATE=lib 定义项目模板是库，而不是应用程序。

CONFIG += staticlib 配置项目为静态库。

TARGET = myStaticLib 定义项目编译后生成的目标文件名称是 myStaticLib。

---

**注意** 静态库项目可以使用 MinGW 或 MSVC 编译器编译，但是项目编译生成的文件与使用的编译器有关。若使用 MSVC 编译，编译后会生成一个库文件 myStaticLib.lib；若使用 MinGW 编译，编译后会生成一个库文件 libmyStaticLib.a。

---

release 和 debug 模式下编译生成的都是相同的文件名，并不会为 debug 版本自动添加一个字母“d”，但是在 release 和 debug 模式下编译应用程序时，需要使用相应版本的库文件。

### 12.3.2 静态链接库的使用

创建一个基于 QMainWindow 的应用程序 LibUser，在项目源程序目录下新建一个 include 目录，根据使用的编译器复制不同的文件。

- 若使用 MSVC 编译器，将静态库项目 myStaticLib 下的 qwdialogpen.h 和 release 版本的 myStaticLib.lib 复制到这个 include 目录下，将 debug 版本的 myStaticLib.lib 更名为 myStaticLibd.lib 复制到这个 include 目录下。

- 若使用 MinGW 编译器，就将 libmyStaticLib.a 和 libmyStaticLibd.a（debug 版本）复制到 include 目录里。

在项目管理目录树里右键单击 LibUser 项目，在快捷菜单里单击“Add Library...”菜单项，在出现的向导对话框里首先选择添加的库类型为“External Library”，在向导第二步设置需要导入的静态库文件（见图 12-9）。

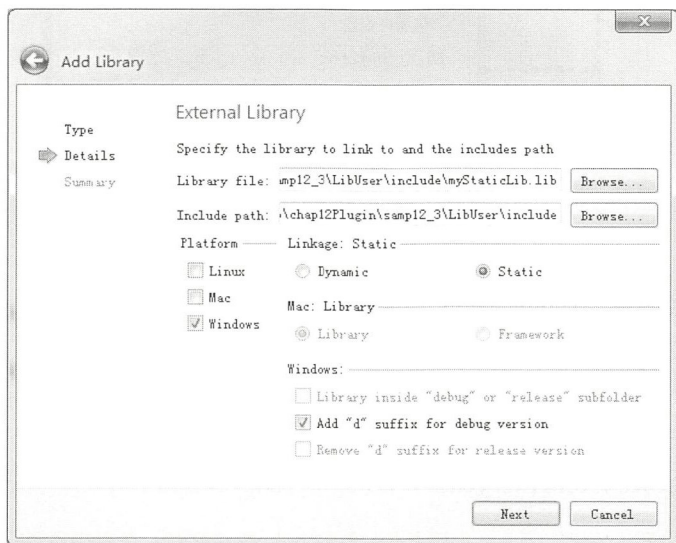


图 12-9 设置添加的静态库信息

首先选择需要导入的库文件 myStaticLib.lib，连接类型里必须选择 Static，因为这是静态库，勾选 Add “d” suffix for debug version，使得在 debug 模式下编译应用程序时将自动调用 debug 版本的库文件 myStaticLibd.lib。

设置完成后，Qt Creator 将自动更改项目配置文件 LibUser.pro，增加以下的内容，主要是设置了包含文件和依赖项的路径，增加了 LIBS 设置。

```
win32:CONFIG(release, debug|release): LIBS += -L$$PWD/include/ -lmyStaticLib
else:win32:CONFIG(debug, debug|release): LIBS += -L$$PWD/include/ -lmyStaticLibd

INCLUDEPATH += $$PWD/include
DEPENDPATH += $$PWD/include

win32-g++:CONFIG(release, debug|release): PRE_TARGETDEPS += $$PWD/include/libmyStaticLib.a
else:win32-g++:CONFIG(debug, debug|release): PRE_TARGETDEPS += $$PWD/include/
libmyStaticLibd.a
else:win32:!win32-g++:CONFIG(release, debug|release): PRE_TARGETDEPS +=
$$PWD/include/ myStaticLib.lib
else:win32:!win32-g++:CONFIG(debug, debug|release): PRE_TARGETDEPS += $$PWD/include/
myStaticLibd.lib
```

编译应用程序 LibUser，使用 MSVC 或 MinGW 编译器，在 release 或 debug 模式下都可以编译，运行程序效果如图 12-10 所示。单击“设置 Pen”按钮可以设置划线的 Pen 属性，并在主窗体

上绘制一个矩形框。



图 12-10 应用程序 LibUser 运行效果

主窗体程序比较简单，MainWindow 类中新增的定义如下：

```
class MainWindow : public QMainWindow
{
private:
    QPen    mPen;
protected:
    void    paintEvent(QPaintEvent *event) Q_DECL_OVERRIDE;
};
```

paintEvent()事件在窗体上绘制一个矩形，使用了QPen类型的私有变量mPen作为绘图的画笔。  
action\_Pen 的响应代码调用静态库里的 QWDialogPen 的静态函数 getPen 设置画笔属性。

```
void MainWindow::paintEvent(QPaintEvent *event)
{ //绘图
    Q_UNUSED(event);
    QPainter painter(this);
    QRect rect(0,0,width(),height()); //viewport 矩形区
    painter.setViewport(rect); //设置 Viewport
    painter.setWindow(0,0,100,50); // 设置窗口大小，逻辑坐标
    painter.setPen(mPen);
    painter.drawRect(10,10,80,30);
}

void MainWindow::on_action_Pen_triggered()
{ //设置 Pen
    bool    ok=false;
    QPen    pen=QWDialogPen::getPen(mPen,ok);
    if (ok)
    { mPen=pen;
      this->repaint();
    }
}
```

本实例将一个可视化设计的对话框 QWDialogPen 封装到一个静态库里，也可以将任何 C++ 类、函数封装到静态库，其实现方法是一样的。

## 12.4 创建和使用共享库

### 12.4.1 创建共享库

除了静态库，Qt 还可以创建共享库，也就是 Windows 平台上的动态链接库。动态链接库项目编译后生成 DLL 文件，DLL 文件在 windows 平台上应用广泛。DLL 文件是在应用程序运行时加载的，不像静态库那样在编译期间就连编到应用程序里。若更新了 DLL 文件版本，只要接口未变，应用程序依然可以调用。

创建共享库项目，单击 Qt Creator 的“File”→“New File or Project”菜单项，在 New File or Project 对话框中选择 Projects 组里的 Library，在右侧的具体类别中再选择 C++ Library，单击“Choose...”按钮后出现如图 12-11 所示的向导对话框。

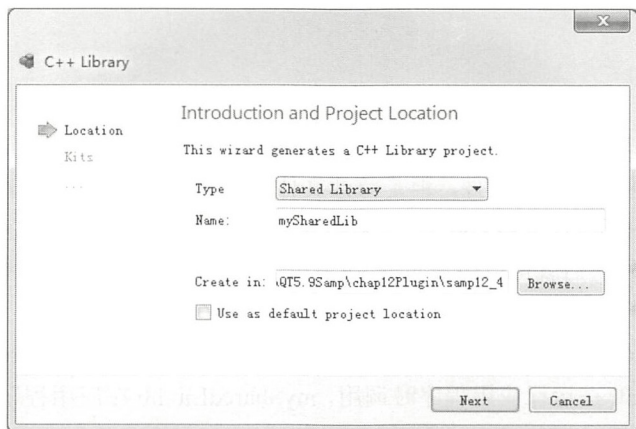


图 12-11 创建共享库向导对话框

在此对话框的 Type 下拉列表框里选择 Shared Library，并给项目命名，例如 mySharedLib，再选择项目保存目录。单击“Next”按钮后选择编译器，下一步选择需要包含的 Qt 模块，再下一步是类定义页面，在其中输入类的名称，这里仍然输入类名称为 QWDialogPen（界面与图 12-8 完全一样），再下一步结束即可。

由向导生成的 mySharedLib 项目包含文件 mySharedLib.pro、qwdialogpen.h 和 qwdialogpen.cpp。此外还有一个特殊的头文件 mysharedlib\_global.h，文件的内容如下：

```
#include <QtCore/qglobal.h>

#ifdef MYSHAREDLIB_LIBRARY
#   define MYSHAREDLIBSHARED_EXPORT Q_DECL_EXPORT
#else
#   define MYSHAREDLIBSHARED_EXPORT Q_DECL_IMPORT
#endif
```

这里定义了符号 MYSHAREDLIBSHARED\_EXPORT 用于替代 Qt 的宏 Q\_DECL\_EXPORT 或



Q\_DECL\_IMPORT。

一个共享库导出给用户使用的类、符号、函数等都需要用宏 Q\_DECL\_EXPORT 来定义导出，一个使用共享库的应用程序需要通过 Q\_DECL\_IMPORT 导入共享库里的可用对象。

在 mySharedLib.pro 文件里增加了符号 MYSHAREDLIB\_LIBRARY 的定义，下面是 mySharedLib.pro 文件的主要内容：

```
Qt    += widgets
TARGET = mySharedLib
TEMPLATE = lib

DEFINES += MYSHAREDLIB_LIBRARY
```

自动生成的 qwdialogpen.h 文件里的内容是对 QWDialogPen 类的定义，在类名称前使用了宏 MYSHAREDLIBSHARED\_EXPORT，定义 QWDialogPen 为一个导出的类。

```
#include "mysharedlib_global.h"
class MYSHAREDLIBSHARED_EXPORT QWDialogPen
{
public:
    QWDialogPen();
};
```

将 12.3 节静态库项目里的文件 qwdialogpen.h、qwdialogpen.cpp 和 qwdialogpen.ui 复制到本项目目录下，覆盖自动生成的初始文件，但是修改文件 qwdialogpen.h 里的类的定义，在类名称前增加 MYSHAREDLIBSHARED\_EXPORT 宏，并加入 mysharedlib\_global.h 的包含语句。

项目的文件准备好之后就可以编译生成 DLL 文件，根据使用的编译器不同，生成的文件有些区别。

- 若使用 MSVC 编译，编译后会生成 mySharedLib.dll 和 mySharedLib.lib 两个文件，mySharedLib.dll 在运行应用程序时调用，mySharedLib.lib 在应用程序隐式调用动态链接库时使用。
- 若使用 MinGW 编译，编译后会生成 mySharedLib.dll 和 libmySharedLib.a 两个文件，mySharedLib.dll 在运行应用程序时调用，libmySharedLib.a 在应用程序隐式调用动态链接库时使用。

采用 debug 和 release 不同模式生成的文件只能当应用程序在 debug 或 release 模式下编译或调用。

## 12.4.2 使用共享库

### 1. 共享库的调用方式

调用动态链接库有两种形式，隐式链接（implicit linking）调用和显式链接（explicit linking）调用。

隐式链接调用是在编译应用程序时，有动态库的 lib 文件（或.a 文件）和 h 头文件，知道 DLL 中有哪些接口类和函数，编译时就隐式地生成必要的链接信息，使用 DLL 中的类或函数时根据 h 头文件中的定义使用即可。应用程序运行时将自动加载 DLL 文件。隐式链接调用主要用于同一种编程软件（如 Qt）生成的代码的共享。



项目编译时,会根据当前是 release 还是 debug 模式,自动添加相应的库文件。这里添加库文件只是使用了动态库的导出定义,而不是将库的实现代码连接到应用程序的可执行文件里。

主窗体类 MainWindow 的功能与 12.3 节的程序完全一致,调用共享库里的类 QWDialogPen 也无需特别说明,只需包含头文件 qwdialogpen.h 即可。

shareLibUser 项目可以用 MinGW 或 MSVC 编译器编译,运行效果与图 12-10 完全一样。

**注意** 必须将动态链接库文件 mySharedLib.dll 复制到可执行文件的目录下,程序才可以正常运行。mySharedLib.dll 的 debug 和 release 版本必须分别用于应用程序的 debug 和 release 版本,否则运行时出错。

使用动态链接库可以很方便地扩展应用程序的功能,但是 DLL 文件需要随应用程序一起发布,并且编译 DLL 和应用程序的 Qt 版本最好保持一致,否则需要考虑二进制兼容问题。

### 3. 显式链接调用共享库

显式链接调用共享库是在应用程序运行时才加载共享库文件,并调用库里的函数的。应用程序编译时无需共享库的任何文件,只需知道函数名和函数的原型即可。所以,这种方式可以调用其他语言编写的 DLL 文件,例如用 Delphi 生成的一个 DLL 文件。

显式链接调用共享库是通过 QLibrary 类实现的。QLibrary 是与平台无关的,用于在运行时载入共享库,一个 QLibrary 对象只对一个共享库进行操作。

一般在 QLibrary 的构造函数中传递一个文件名,可以是带路径的绝对文件名,也可以是不带后缀的单独文件名。QLibrary 会根据运行的平台自动查找不同后缀的共享库文件,例如 Unix 上是“.so”,Mac 上是“.dylib”,Windows 上是“.dll”。

作为示例,用 Delphi 编写一个 DLL 项目,生成一个 DelphiDLL.dll 文件,这个文件里只有一个函数,函数的原型为:

```
function triple(N:integer):integer;
```

它会计算传递参数 N 的 3 倍值并返回。

在 Qt Creator 里创建一个基于 QMainWindow 的应用程序 DelphiDLLUser,设计一个简单的界面,运行时如图 12-13 所示。单击按钮时将根据输入,调用动态链接库 DelphiDLL.dll 里的 triple() 函数,计算结果并显示在输出编辑框里。

按钮的槽函数代码如下:

```
void MainWindow::on_pushButton_clicked()
{
    QLibrary myLib("DelphiDLL");
    if (myLib.isLoaded())
        QMessageBox::information(this, "信息", "DelphiDLL.DLL 已经被载入,第 1 处");
    typedef int (*FunDef)(int);
    FunDef myTriple = (FunDef) myLib.resolve("triple"); //解析 DLL 中的函数
    int V=myTriple(ui->spinInput->value()); //调用函数
    ui->spinOutput->setValue(V);
    if (myLib.isLoaded())
        QMessageBox::information(this, "信息", "DelphiDLL.DLL 已经被载入,第 2 处");
}
```



图 12-13 DelphiDLLUser 运行界面

在定义 QLibrary 对象实例 myLib 时传递了共享库文件名“DelphiDLL”，这里不需要给出后缀名。DelphiDLL.dll 文件必须在应用程序同一目录、系统目录或可搜索目录下。

QLibrary 有几个函数用于 DLL 文件的载入与卸载：load()用于手动载入 DLL 文件到内存里，一般无需手工调用此函数，在 DLL 里的函数第一次被使用时 QLibrary 会自动调用此函数；isLoading()用于判断 DLL 是否已经被载入内存；unload()用于将 DLL 从内存中卸载。

一个动态链接库在内存里只能有一个实例，也就是即使有多处调用了这个动态链接库里的函数，它也只会被载入一次，如果不是所有的实例都使用 unload()卸载它，那么它会在应用程序退出时才卸载。

在槽函数 on\_pushButton\_clicked()的代码里，有两处 QMessageBox 显示信息。在运行应用程序，第一次单击按钮时，只有第 2 处信息框显示，说明声明了 QLibrary 对象后，动态链接库没有立即被载入内存；第二次单击按钮时，两处信息框会先后显示，说明动态链接库上次载入内存后，还在内存里。

显式调用动态链接库里的函数，需要声明函数原型的类型，即：

```
typedef int (*FunDef)(int);
```

然后使用 QLibrary 的 resolve()函数解析需要调用的函数。

```
FunDef myTriple = (FunDef) myLib.resolve("triple");
```

这样就定义了一个函数 myTriple，用于实现 DLL 文件里的函数“triple”的功能，当然重新声明的函数名称可以和 DLL 里的函数名称完全相同。

如果 DelphiDLL.dll 文件没有复制到应用程序目录下，则编译和启动应用程序都不会出错，只有单击按钮调用 DLL 里的函数时才会出错。所以，要使应用程序正常运行，需要将 DelphiDLL.dll 文件复制到应用程序目录下。



# 多线程

一个应用程序一般只有一个线程，一个线程内的操作是顺序执行的，如果有某个比较消耗时间的计算或操作，比如网络通信中的文件传输，在一个线程内操作时，用户界面就可能会冻结而不能及时响应。这种情况下，可以创建一个单独的线程来执行比较消耗时间的操作，并与主线程之间处理好同步与数据交互，这就是多线程应用程序。

Qt 为多线程操作提供了完整的支持。QThread 是线程类，是实现多线程操作的核心类，一般从 QThread 继承定义自己的线程类。线程之间的同步是其交互的主要问题，Qt 提供了 QMutex、QMutexLocker、QReadWriteLock、QwaitCondition、QSemaphore 等多种类用于实现线程之间的同步。Qt 还有 Qt Concurrent 模块，提供一些高级的 API 实现多线程编程而无需使用 QMutex、QwaitCondition 和 QSemaphore 等基础操作。使用 Qt Concurrent 实现的多线程程序可以自动根据处理器内核个数调整线程个数。

本章主要介绍用 QThread 实现多线程编程的方法，以及用 QMutex、QWaitCondition、QSemaphore 等实现线程同步的方法。

## 13.1 QThread 创建多线程程序

### 13.1.1 QThread 类功能简介

QThread 类提供不依赖于平台的管理线程的方法。一个 QThread 类的对象管理一个线程，一般从 QThread 继承一个自定义类，并重定义虚函数 run()，在 run()函数里实现线程需要完成的任务。

将应用程序的线程称为主线程，额外创建的线程称为工作线程。一般在主线程里创建工作线程，并调用 start()开始执行工作线程的任务。start()会在内部调用 run()函数，进入工作线程的事件循环，在 run()函数里调用 exit()或 quit()可以结束线程的事件循环，或在主线程里调用 terminate()强制结束线程。

QThread 类的主要接口函数、信号和槽函数见表 13-1。

表 13-1 Qthread 类的主要接口

类型	函数	功能
公共函数	bool isFinished()	线程是否结束
	bool isRunning()	线程是否正在运行
	Priority priority()	返回线程的优先级

续表

类型	函数	功能
公共函数	void setPriority (Priority priority)	设置线程的优先级
	void exit(int returnCode = 0)	退出线程的事件循环，退出码为 returnCode，0 表示成功退出；否则表示有错误
	bool wait(unsigned long time )	阻止线程执行，直到线程结束（从 run()函数返回），或等待时间超过 time 毫秒
公共槽函数	void quit()	退出线程的事件循环，并返回代码 0，等效于 exit(0)
	void start(Priority priority )	内部调用 run()开始执行线程，操作系统根据 priority 参数进行调度
	void terminate()	终止线程的运行，但不是立即结束线程，而是等待操作系统结束线程。使用 terminate()之后应使用 wait()
信号	void finished()	在线程就要结束时发射此信号
	void started()	在线程开始执行、run()函数被调用之前发射此信号
静态公共成员	int idealThreadCount()	返回系统上能运行的线程的理想个数
	void msleep(unsigned long msecs)	强制当前线程休眠 msecs 毫秒
	void sleep(unsigned long secs)	强制当前线程休眠 secs 秒
	void usleep(unsigned long usecs)	强制当前线程休眠 usecs 微秒
保护函数	virtual void run()	start()调用 run()函数开始线程任务的执行，所以在 run()函数里实现线程的任务功能
	int exec()	由 run()函数调用，进入线程的事件循环，等待 exit()退出

QThread 是 QObject 的子类，所以可以使用信号与槽机制。QThread 自身定义了 started()和 finished()两个信号，started()信号在线程开始执行之前发射，也就是在 run()函数被调用之前，finished()信号在线程就要结束时发射。

### 13.1.2 掷骰子的线程 QDiceThread

作为实例，定义一个掷骰子的线程类 QDiceThread，类的声明部分如下：

```
#include <Qthread>
class QDiceThread : public Qthread
{
    Q_OBJECT
private:
    int    m_seq=0;           //掷骰子次数序号
    int    m_diceValue;       //骰子点数
    bool   m_Paused=true;     //暂停
    bool   m_stop=false;      //停止
protected:
    void   run() Q_DECL_OVERRIDE; //线程任务
public:
    QDiceThread();
    void   diceBegin();         //掷一次骰子
    void   dicePause();         //暂停
    void   stopThread();        //结束线程
signals:
    void   newValue(int seq,int diceValue); //产生新点数的信号
};
```

重载虚函数 run()，在此函数里完成线程的主要任务。

自定义 diceBegin()、dicePause()、stopThread() 3 个公共函数用于线程控制，这 3 个函数由主

线程调用。

定义了一个信号 `newValue(int seq,int diceValue)` 用于在掷一次骰子得到新的点数之后发射此信号，由主线程的槽函数响应以获取值。

`QDiceThread` 类的实现代码如下：

```
#include "qdicethread.h"
#include <Qtime>
QDiceThread::QDiceThread()
{ //构造函数
}
void QDiceThread::diceBegin()
{ //开始掷骰子
    m_Paused=false;
}
void QDiceThread::dicePause()
{ //暂停掷骰子
    m_Paused=true;
}
void QDiceThread::stopThread()
{ //停止线程
    m_stop=true;
}
void QDiceThread::run()
{ //线程任务
    m_stop=false; //启动线程时令 m_stop=false
    m_seq=0; //掷骰子次数
    qsrand(Qtime::currentTime().msec()); //随机数初始化, qsrnd 是线程安全的
    while(!m_stop) //循环主体
    {
        if (!m_Paused)
        {
            m_diceValue=qrnd(); //获取随机数
            m_diceValue=(m_diceValue % 6)+1;
            m_seq++;
            emit newValue(m_seq,m_diceValue); //发射信号
        }
        msleep(500); //线程休眠 500ms
    }
    quit(); //相当于 exit(0), 退出线程的事件循环
}
```

其中，`run()`是线程任务的实现部分，线程开始就执行 `run()`函数。`run()`函数一般是事件循环过程，根据各种条件或事件处理各种任务。当 `run()`函数退出时，线程的事件循环就结束了。

在 `run()`函数里，初始化变量 `m_stop` 和 `m_seq`，用 `qsrand()`函数对随机数种子初始化。`run()`函数的主体是一个 `while` 循环，在主线程调用 `stopThread()`函数使 `m_stop` 为 `true`，才会退出 `while` 循环，调用 `quit()`之后结束线程。

在 `while` 循环体内，又根据 `m_Paused` 判断当前是否需要掷骰子，如果需要掷骰子，则用随机函数生成一次骰子的点数 `m_diceValue`，然后发射信号 `newValue()`，将 `m_seq` 和 `m_diceValue` 作为信号参数传递出去。主线程可以设计槽函数与此信号关联，获取这两个值并进行显示。

### 13.1.3 掷骰子的多线程应用程序

使用 QDiceThread 类，设计一个应用程序 samp13\_1，程序运行界面如图 13-1 所示。

窗体上方的几个按钮用于控制线程的启动与停止，控制开始与暂停掷骰子。中间的文本框显示次数和点数，右边根据点数显示资源文件里面的一个图片，图片存储在项目的资源文件里。下方的一个标签根据 QDiceThread 的 started() 和 finished() 两个信号显示线程的状态。

窗口类是从 QDialog 继承的类 Dialog，其类定义如下（省略了按钮槽函数的定义）：

```
class Dialog : public QDialog
{
    Q_OBJECT
private:
    QDiceThread threadA;
protected:
    void closeEvent(QCloseEvent *event);
public:
    explicit Dialog(QWidget *parent = 0);
    ~Dialog();
private slots:
    //自定义槽函数
    void onthreadA_started();
    void onthreadA_finished();
    void onthreadA_newValue(int seq, int diceValue);
private:
    Ui::Dialog *ui;
};
```

这里定义了一个 QDiceThread 类型的变量 threadA，重定义了 closeEvent() 事件，自定义了 3 个槽函数。

Dialog 类的构造函数代码如下：

```
Dialog::Dialog(QWidget *parent) : QDialog(parent), ui(new Ui::Dialog)
{
    //构造函数
    ui->setupUi(this);
    connect(&threadA, SIGNAL(started()), this, SLOT(onthreadA_started()));
    connect(&threadA, SIGNAL(finished()), this, SLOT(onthreadA_finished()));
    connect(&threadA, SIGNAL(newValue(int, int)),
            this, SLOT(onthreadA_newValue(int, int)));
}
```

构造函数主要是将 threadA 的 3 个信号与 Dialog 自定义的 3 个槽函数相关联，这 3 个槽函数的代码如下：

```
void Dialog::onthreadA_started()
{
    //线程的 started() 信号的响应槽函数
    ui->LabA->setText("Thread 状态: thread started");
}
```

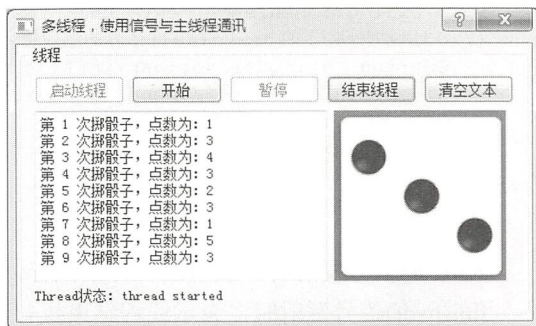


图 13-1 掷骰子多线程应用程序 samp13\_1 运行界面



```

}
void Dialog::onthreadA_finished()
{//线程的 finished() 信号的响应槽函数
    ui->LabA->setText("Thread 状态: thread finished");
}
void Dialog::onthreadA_newValue(int seq,int diceValue)
{//QDiceThread 的 newValue() 信号的响应槽函数, 显示骰子次数和点数
    QString str=QString::asprintf("第 %d 次掷骰子, 点数为: %d",seq,diceValue);
    ui->plainTextEdit->appendPlainText(str);
    QPixmap pic; //图片显示
    QString filename=QString::asprintf(":/dice/images/d%d.jpg",diceValue);
    pic.load(filename);
    ui->LabPic->setPixmap(pic);
}

```

started()信号发射时, 表示线程开始执行, 在标签里显示状态文字。

finished()信号发射时, 表示线程结束执行, 在标签里显示状态文字。

newValue()是 QDiceThread 定义的信号, 在掷一次骰子获得新的点数后发射, 将掷骰子的次数和点数传递过来。槽函数 onthreadA\_newValue()获取这两个值并显示在文本框里, 再根据点数从资源文件里获取相应的图片并显示。

窗口上 5 个按钮的代码如下:

```

void Dialog::on_btnStartThread_clicked()
{//启动线程 按钮
    threadA.start();
    ui->btnStartThread->setEnabled(false);
    ui->btnStopThread->setEnabled(true);
    ui->btnDiceBegin->setEnabled(true);
    ui->btnDiceEnd->setEnabled(false);
}
void Dialog::on_btnStopThread_clicked()
{//结束线程 按钮
    threadA.stopThread();//结束线程的 run() 函数执行
    threadA.wait();
    ui->btnStartThread->setEnabled(true);
    ui->btnStopThread->setEnabled(false);
    ui->btnDiceBegin->setEnabled(false);
    ui->btnDiceEnd->setEnabled(false);
}
void Dialog::on_btnDiceBegin_clicked()
{//开始 掷骰子按钮
    threadA.diceBegin();
    ui->btnDiceBegin->setEnabled(false);
    ui->btnDiceEnd->setEnabled(true);
}
void Dialog::on_btnDiceEnd_clicked()
{//暂停 掷骰子按钮
    threadA.dicePause();
    ui->btnDiceBegin->setEnabled(true);
    ui->btnDiceEnd->setEnabled(false);
}
void Dialog::on_btnClear_clicked()
{//清空文本 按钮
    ui->plainTextEdit->clear();
}

```

```
}
```

“启动线程”按钮调用线程的 `start()` 函数，`start()` 函数会内部调用 `run()` 函数开始线程任务的执行。`run()` 函数将内部变量 `m_Paused` 初始化为 `true`，所以，启动线程后并不会立即开始掷骰子。

“开始”按钮调用 `diceBegin()` 函数，使 `threadA` 线程内部变量 `m_Paused` 变为 `false`，那么 `run()` 函数里就开始每隔 500 毫秒产生一次骰子点数，并发射信号 `newValue()`。

“暂停”按钮调用 `dicePause()` 函数，使 `threadA` 线程内部变量 `m_Paused` 变为 `true`，`run()` 函数里不再掷骰子，但是 `run()` 函数并没有结束，也就是线程并没有结束。

“结束线程”按钮调用 `stopThread()` 函数，使 `threadA` 线程内部的 `m_stop` 变为 `true`，`run()` 函数体的 `while` 循环结束，执行 `quit()` 后线程结束。所以，线程结束就是 `run()` 函数执行退出。

重载 `closeEvent()` 事件，在窗口关闭时确保线程被停止，代码如下：

```
void Dialog::closeEvent(QCloseEvent *event)
{ //窗口关闭事件，必须结束线程
    if (threadA.isRunning())
    {
        threadA.stopThread();
        threadA.wait();
    }
    event->accept();
}
```

## 13.2 线程同步

### 13.2.1 线程同步的概念

在多线程应用程序中，由于多个线程的存在，线程之间可能需要访问同一个变量，或一个线程需要等待另外一个线程完成某个操作后才产生相应的动作。例如，在上一节的实例 `samp13_1` 中，工作线程产生随机的骰子点数，主线程读取骰子点数并显示，主线程需要等待工作线程产生一个新的骰子点数后再读取数据。实例 `samp13_1` 中使用了信号与槽的机制，在产生新的骰子数之后通过信号通知主线程读取新的数据。

如果不使用信号与槽机制，`QDiceThread` 的 `run()` 函数变为如下的代码：

```
void QDiceThread::run()
{
    m_stop=false;//启动线程时令 m_stop=false
    m_seq=0;
    qsrand(Qtime::currentTime().msec());//随机数初始化，qsrand 是线程安全的
    while(!m_stop)//循环主体
    {
        if (!m_paused)
        {
            m_diceValue=qrand(); //获取随机数
            m_diceValue=(m_diceValue % 6)+1;
            m_seq++;
        }
        msleep(500); //线程休眠
    }
}
```

```
    }
}
```

那么, QDiceThread 需要定义公共函数, 返回 m\_diceValue 的值, 如:

```
int QDiceThread::diceValue () { return m_diceValue; }
```

以便在主线程中调用此函数读取骰子的点数。

由于没有信号与槽的关联 (信号与槽的关系类似于硬件的中断与中断处理函数), 主线程只能采用不断查询的方式主动查询是否有新数据, 并读取它。但是在主线程调用 diceValue() 读取骰子点数时, 工作线程可能正在执行 run() 函数里修改 m\_diceValue 值的语句, 即:

```
m_diceValue=qrand(); //获取随机数
m_diceValue=(m_diceValue % 6)+1;
m_seq++;
```

而且这几条语句计算量大, 需要执行较长时间。执行这两条语句时不希望被主线程调用的 diceValue() 函数中断, 如果中断, 则主线程得到的可能是错误的值。

这种情况下, 这样的代码段是希望被保护起来的, 在执行过程中不能被其他线程打断, 以保证计算结果的完整性, 这就是线程同步的概念。

在 Qt 中, 有多个类可以实现线程同步的功能, 包括 QMutex、QMutexLocker、QReadWriteLock、QReadLocker、QWriteLocker、QWaitCondition 和 QSemaphore。下面将分别介绍这些类的用法。

### 13.2.2 基于互斥量的线程同步

QMutex 和 QMutexLocker 是基于互斥量的线程同步类, QMutex 定义的实例是一个互斥量, QMutex 主要提供 3 个函数。

- lock(): 锁定互斥量, 如果另外一个线程锁定了这个互斥量, 它将阻塞执行直到其他线程解锁这个互斥量。
- unlock(): 解锁一个互斥量, 需要与 lock() 配对使用。
- tryLock(): 试图锁定一个互斥量, 如果成功锁定就返回 true; 如果其他线程已经锁定了这个互斥量, 就返回 false, 但不阻塞程序执行。

使用互斥量, 对 QDiceThread 类重新定义, 不采用信号与槽机制, 而是提供一个函数用于主线程读取数据。更改后的 QDiceThread 类定义如下:

```
class QDiceThread : public QThread
{
    Q_OBJECT
private:
    QMutex    mutex; //互斥量
    int       m_seq=0; //序号
    int       m_diceValue;
    bool      m_paused=true;
    bool      m_stop=false; //停止线程
protected:
    void      run() Q_DECL_OVERRIDE;
public:
    QDiceThread();
```

```

void    diceBegin();//掷一次骰子
void    diceEnd();
void    stopThread();
bool    readValue(int *seq, int *diceValue); //用于主线程读取数据的函数
};

```

QDiceThread 类里用 QMutex 类定义了一个互斥量变量 mutex。

定义了函数 readValue(), 用于外部线程读取掷骰子的次数和点数, 传递参数采用指针变量, 以便一次读取两个数据。

下面是 QDiceThread 类中关键的 run()和 readValue()函数的实现代码。

```

void QDiceThread::run()
{
    m_stop=false;
    m_seq=0;
    qsrand(Qtime::currentTime().msec()); //随机数初始化
    while(!m_stop) //循环主体
    {
        if (!m_paused)
        {
            mutex.lock();
            m_diceValue=qrand(); //获取随机数
            m_diceValue=(m_diceValue % 6)+1;
            m_seq++;
            mutex.unlock();
        }
        msleep(500); //线程休眠
    }
}

bool QDiceThread::readValue(int *seq, int *diceValue)
{
    if (mutex.tryLock())
    {
        *seq=m_seq;
        *diceValue=m_diceValue;
        mutex.unlock();
        return true;
    }
    else
        return false;
}

```

在 run()函数中, 对重新计算骰子点数和掷骰子次数的 3 行代码用互斥量 mutex 的 lock()和 unlock()进行了保护, 这部分代码的执行就不会被其他线程中断。注意, lock()与 unlock()必须配对使用。

在 readValue()函数中, 用互斥量 mutex 的 tryLock()和 unlock()进行了保护。如果 tryLock()成功锁定互斥量, 读取数值的两行代码执行时不会被中断, 执行完后解锁; 如果 tryLock()锁定失败, 函数就立即返回, 而不会等待。

原理上, 对于两个或多个线程可能会同时读或写的变量应该使用互斥量进行保护, 例如 QDiceThread 中的变量 m\_stop 和 m\_paused, 在 run()函数中读取这两个变量, 要在 diceBegin()、diceEnd()和 stopThread()函数里修改这些值, 但是这 3 个函数都只有一条赋值语句, 可以认为是原



子操作，所以，可以不用锁定保护。

定义的互斥量 `mutex` 相当于一个标牌，可以这样来理解互斥量：列车上的卫生间一次只能进一个人，当一个人尝试进入卫生间就是 `lock()`，如果有人占用，他就只能等待；等里面的人出来，腾出了卫生间是 `unlock()`，这个等待的人才可以进入并且锁住卫生间的门，就是 `lock()`，使用完卫生间之后他再出来时就是 `unlock()`。

`QMutex` 需要配对使用 `lock()`和 `unlock()`来实现代码段的保护，在一些逻辑复杂的代码段或可能发生异常的代码中，配对就可能出错。

`QMutexLocker` 是另外一个简化了互斥量处理的类。`QMutexLocker` 的构造函数接受一个互斥量作为参数并将其锁定，`QMutexLocker` 的析构函数则将此互斥量解锁，所以在 `QMutexLocker` 实例变量的生存期内的代码段得到保护，自动进行互斥量的锁定和解锁。例如，`QDiceThread` 的 `run()` 函数的代码可以改写如下：

```
void QDiceThread::run()
{
    m_stop=false;
    m_seq=0;
    qsrand(Qtime::currentTime().msec()); //随机数初始化
    while(!m_stop) //循环主体
    {
        if (!m_paused)
        {
            QMutexLocker Locker(&mutex);
            m_diceValue=grand(); //获取随机数
            m_diceValue=(m_diceValue % 6)+1;
            m_seq++;
        }
        msleep(500); //线程休眠
    }
}
```

这样定义的 `QDiceThread` 类，在主程序中只能调用其 `readValue()`函数来不断读取数值。实例 `samp13_2` 采用 `QMutex` 进行线程同步，实例 `samp13_3` 采用 `QMutex` 和 `QMutexLocker` 进行线程同步，其界面与 `samp13_1` 完全相同，只是增加了定时器，用于定时主动去读取掷骰子线程的数值。

实例程序 `samp13_2` 的 `Dialog` 类的主要定义如下（省略了一些系统生成的声明）：

```
class Dialog : public QDialog
{
private:
    int mSeq, mDiceValue;
    QDiceThread threadA;
    QTimer mTimer; //定时器
public:
    explicit Dialog(QWidget *parent = 0);
private slots:
    void onthreadA_started();
    void onthreadA_finished();
    void onTimeOut(); //定期器处理槽函数
};
```

主要是增加了一个定时器 `mTimer` 和其时间溢出响应槽函数 `onTimeOut()`，在 `Dialog` 的构造函数

数中将 mTimer 的 timeout 信号与此槽函数关联。

```
connect(&mTimer, SIGNAL(timeout()), this, SLOT(onTimeOut()));
```

onTimeOut()函数的主要功能是调用 threadA 的 readValue()函数读取数值。定时器的定时周期设置为 100ms, 小于 threadA 产生一次新数据的周期 (500ms), 所以可能读出旧的数据, 通过存储的掷骰子的次数与读取的掷骰子次数是否不同, 判断是否为新数据。onTimeOut()函数的代码如下:

```
void Dialog::onTimeOut()
{ //定时器溢出处理槽函数
    int tmpSeq=0,tmpValue=0;
    bool valid=threadA.readValue(&tmpSeq,&tmpValue); //读取数值
    if (valid && (tmpSeq!=mSeq)) //有效, 并且是新数据
    {
        mSeq=tmpSeq;
        mDiceValue=tmpValue;
        QString str=QString::asprintf("第 %d 次掷骰子, 点数为: %d", mSeq,mDiceValue);
        ui->plainTextEdit->appendPlainText(str);
        QPixmap pic;
        QString filename=QString::asprintf(":/dice/images/d%d.jpg",mDiceValue);
        pic.load(filename);
        ui->LabPic->setPixmap(pic);
    }
}
```

窗口上几个按钮的代码如下 (省略了按钮使能控制的代码):

```
void Dialog::on_btnStartThread_clicked()
{ //启动线程
    mSeq=0;
    threadA.start();
}
void Dialog::on_btnStopThread_clicked()
{ //结束线程
    threadA.stopThread(); //结束线程的 run() 函数执行
    threadA.wait();
}
void Dialog::on_btnDiceBegin_clicked()
{ //开始掷骰子
    threadA.diceBegin();
    mTimer.start(100); //定时器 100ms 读取一次数据
}
void Dialog::on_btnDiceEnd_clicked()
{ //暂停掷骰子
    threadA.diceEnd();
    mTimer.stop(); //定时器暂停
}
```

实例 samp13\_2 和 samp13\_3 实现的效果与实例 samp13\_1 相同, 只是实现的方式不同。

### 13.2.3 基于 QReadWriteLock 的线程同步

使用互斥量时存在一个问题: 每次只能有一个线程获得互斥量的权限。如果在一个程序中有多个线程读取某个变量, 使用互斥量时也必须排队。而实际上若只是读取一个变量, 是可以让多

个线程同时访问的，这样互斥量就会降低程序的性能。

例如，假设有一个数据采集程序，一个线程负责采集数据到缓冲区，一个线程负责读取缓冲区的数据并显示，另一个线程负责读取缓冲区的数据并保存到文件，示意代码如下：

```
int    buffer[100];
QMutex mutex;
void   threadDAQ::run()
{
    ...
    mutex.lock();
    get_data_and_write_in_buffer();    //数据写入 buffer
    mutex.unlock();
    ...
}
void   threadShow::run()
{
    ...
    mutex.lock();
    show_buffer();    //读取 buffer 里的数据并显示
    mutex.unlock();
    ...
}
void   threadSaveFile::run()
{
    ...
    mutex.lock();
    Save_buffer_toFile();    //读取 buffer 里的数据并保存到文件
    mutex.unlock();
    ...
}
```

数据缓冲区 `buffer` 和互斥量 `mutex` 都是全局变量，线程 `threadDAQ` 将数据写到 `buffer`，线程 `threadShow` 和 `threadSaveFile` 只是读取 `buffer`，但是因为使用互斥量，这 3 个线程任何时候都只能有一个线程可以访问 `buffer`。而实际上，`threadShow` 和 `threadSaveFile` 都只是读取 `buffer` 的数据，它们同时访问 `buffer` 是不会发生冲突的。

Qt 提供了 `QReadWriteLock` 类，它是基于读或写的模式进行代码段锁定的，在多个线程读写一个共享数据时，可以解决上面所说的互斥量存在的问题。

`QReadWriteLock` 以读或写锁定的同步方法允许以读或写的方式保护一段代码，它可以允许多个线程以只读方式同步访问资源，但是只要有一个线程在以写方式访问资源时，其他线程就必须等待直到写操作结束。

`QReadWriteLock` 提供以下几个主要的函数：

- `lockForRead()`，以只读方式锁定资源，如果有其他线程以写入方式锁定，这个函数会阻塞；
- `lockForWrite()`，以写入方式锁定资源，如果本线程或其他线程以读或写模式锁定资源，这个函数就阻塞；
- `unlock()`，解锁；
- `tryLockForRead()`，是 `lockForRead()` 的非阻塞版本；
- `tryLockForWrite()`，是 `lockForWrite()` 的非阻塞版本。

使用 `QReadWriteLock`，上面的三线程代码可以改写为如下的形式：

```
int    buffer[100];
```

```

QReadWriteLock Lock;
void threadDAQ::run()
{
    ...
    Lock.lockForWrite();
    get_data_and_write_in_buffer(); //数据写入 buffer
    Lock.unlock();
    ...
}
void threadShow::run()
{
    ...
    Lock.lockForRead();
    show_buffer(); //读取 buffer 里的数据并显示
    Lock.unlock();
    ...
}
void threadSaveFile::run()
{
    ...
    Lock.lockForRead();
    Save_buffer_toFile(); //读取 buffer 里的数据并保存到文件
    Lock.unlock();
    ...
}

```

这样，如果 threadDAQ 没有以 lockForWrite() 锁定 Lock，threadShow 和 threadSaveFile 可以同时访问 buffer，否则 threadShow 和 threadSaveFile 都被阻塞；如果 threadShow 和 threadSaveFile 都没有锁定，那么 threadDAQ 能以写入方式锁定，否则 threadDAQ 就被阻塞。

QReadLocker 和 QWriteLocker 是 QReadWriteLock 的简便形式，如同 QMutexLocker 是 QMutex 的简便版本一样，无需与 unlock() 配对使用。使用 QReadLocker 和 QWriteLocker，则上面的代码改写为：

```

int buffer[100];
QReadWriteLock Lock;
void threadDAQ::run()
{
    ...
    QWriteLocker Locker(&Lock);
    get_data_and_write_in_buffer(); //数据写入 buffer
    ...
}
void threadShow::run()
{
    ...
    QReadLocker Locker(&Lock);
    show_buffer(); //读取 buffer 里的数据并显示
    ...
}
void threadSaveFile::run()
{
    ...
    QReadLocker Locker(&Lock);
    Save_buffer_toFile(); //读取 buffer 里的数据并保存到文件
    ...
}

```

### 13.2.4 基于 QWaitCondition 的线程同步

在多线程的程序中，多个线程之间的同步实际上就是它们之间的协调问题。例如上一小节讲



到的3个线程的例子中,假设 threadDAQ 写满一个缓冲区之后, threadShow 和 threadSaveFile 才能对缓冲区进行读操作。前面采用的互斥量和基于 QReadWriteLock 的方法都是对资源的锁定和解锁,避免同时访问资源时发生冲突。在一个线程解锁资源后,不能及时通知其他线程。

QWaitCondition 提供了另外一种改进的线程同步方法, QWaitCondition 与 QMutex 结合,可以使一个线程在满足一定条件时通知其他多个线程,使它们及时作出响应,这样比只使用互斥量效率要高一些。例如, threadDAQ 在写满一个缓冲区之后,及时通知 threadShow 和 threadSaveFile,使它们可以及时读取缓冲区数据。

QWaitCondition 提供如下一些函数:

- wait(QMutex \*lockedMutex), 解锁互斥量 lockedMutex, 并阻塞等待唤醒条件, 被唤醒后锁定 lockedMutex 并退出函数;
- wakeAll(), 唤醒所有处于等待状态的线程, 线程唤醒的顺序不确定, 由操作系统的调度策略决定;
- wakeOne(), 唤醒一个处于等待状态的线程, 唤醒哪个线程不确定, 由操作系统的调度策略决定。

QWaitCondition 一般用于“生产者/消费者”(producer/consumer)模型中。“生产者”产生数据,“消费者”使用数据,前述的数据采集、显示与存储的三线程例子就适用这种模型。

创建实例程序 samp13\_4, 将掷骰子的程序修改为 producer/consumer 模型, 一个线程类 QThreadProducer 专门负责掷骰子产生点数; 一个线程类 QThreadConsumer 专门及时读取数据, 并送给主线程进行显示。这两个类定义在一个文件 qmythread.h 里, 定义代码如下:

```
class QThreadProducer : public Qthread
{
    Q_OBJECT
private:
    bool    m_stop=false; //停止线程
protected:
    void    run() Q_DECL_OVERRIDE;
public:
    QThreadProducer();
    void    stopThread();
};

class QThreadConsumer : public Qthread
{
    Q_OBJECT
private:
    bool    m_stop=false; //停止线程
protected:
    void    run() Q_DECL_OVERRIDE;
public:
    QThreadConsumer();
    void    stopThread();
signals:
    void    newValue(int seq,int diceValue);
};
```

QThreadProducer 用于掷骰子, 但是去掉了开始和暂停的功能, 线程一启动就连续地掷骰子。QThreadConsumer 用于读取掷骰子的次数和点数, 并用发射信号方式把数据传递出去。这两个类

的实现代码在一个文件 `qmythread.cpp` 里，下面是这两个类的实现代码的主要部分：

```
QMutex mutex;
QWaitCondition newdataAvailable;
int seq=0; //序号
int diceValue;
void QthreadProducer::run()
{
    m_stop=false;
    seq=0;
    qsrand(Qtime::currentTime().msec()); //随机数初始化
    while(!m_stop) //循环主体
    {
        mutex.lock();
        diceValue=grand(); //获取随机数
        diceValue=(diceValue % 6)+1;
        seq++;
        mutex.unlock();
        newdataAvailable.wakeAll(); //唤醒所有线程，有新数据了
        msleep(500); //线程休眠
    }
}

void QthreadConsumer::run()
{
    m_stop=false;
    while(!m_stop) //循环主体
    {
        mutex.lock();
        newdataAvailable.wait(&mutex); //先解锁 mutex，使其他线程可以使用 mutex
        emit newValue(seq,diceValue);
        mutex.unlock();
    }
}
```

掷骰子的次数和点数的变量定义为共享变量，这样两个线程都可以访问。定义了互斥量 `mutex`，定义了 `QWaitCondition` 实例 `newdataAvailable`，表示有新数据可用了。

`QThreadProducer::run()` 函数负责每隔 500 毫秒掷骰子产生一次数据，新数据产生后通过等待条件唤醒所有等待的线程，即：

```
newdataAvailable.wakeAll();
```

`QThreadConsumer::run()` 函数中的 `while` 循环，首先需要将互斥量锁定，再执行下面的一条语句：

```
newdataAvailable.wait(&mutex);
```

这条语句以 `mutex` 作为输入参数，内部会首先解锁 `mutex`，使其他线程可以使用 `mutex`，`newdataAvailable` 进入等待状态。当 `QThreadProducer` 产生新数据使用 `newdataAvailable.wakeAll()` 唤醒所有线程后，`newdataAvailable.wait(&mutex)` 会再次锁定 `mutex`，然后退出阻塞状态，以执行后面的语句。

所以，使用 `QWaitCondition` 可以使 `QThreadConsumer` 线程的执行过程进入等待状态。在 `QThreadProducer` 线程满足条件后，唤醒 `QThreadConsumer` 线程及时退出等待状态，继续执行后面的程序。

使用 QThreadProducer 和 QThreadConsumer 实现掷骰子的实例程序 samp13\_4 运行时界面如图 13-2 所示, 与实例 samp13\_1 的运行界面类似, 只是取消了开始和暂停掷骰子的按钮, 下方的状态标签显示了两个线程的状态。

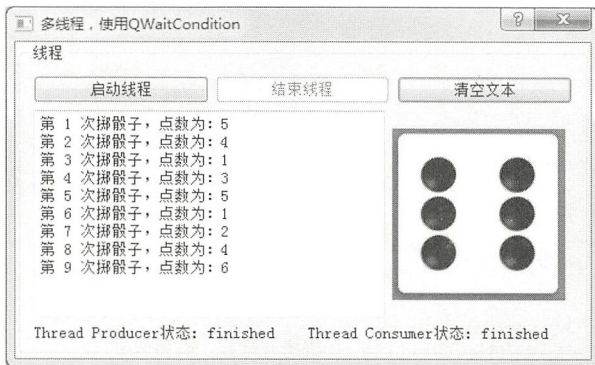


图 13-2 使用 QWaitCondition 的线程同步实例程序 samp13\_4 运行界面

窗口的 Dialog 类的定义如下 (省略了按钮槽函数等一些不重要的部分):

```
class Dialog : public QDialog
{
    Q_OBJECT
private:
    QthreadProducer    threadProducer;
    QthreadConsumer    threadConsumer;
protected:
    void    closeEvent(QCloseEvent *event);
public:
    explicit Dialog(QWidget *parent = 0);
private slots:
    void    onthreadA_started();
    void    onthreadA_finished();
    void    onthreadB_started();
    void    onthreadB_finished();
    void    onthreadB_newValue(int seq, int diceValue);
};
```

这里主要是定义了两个线程的实例, 并定义了几个自定义槽函数。采用信号与槽的方式与 threadConsumer 建立通信并获取数据。Dialog 的构造函数主要完成信号与槽函数的关联, 5 个自定义槽函数的代码与实例 samp13\_1 的相同或相似, 这几个函数的代码不再详细列出。

“启动线程”按钮的代码如下:

```
void Dialog::on_btnStartThread_clicked()
{
    //启动线程
    threadConsumer.start();
    threadProducer.start();
    ui->btnStartThread->setEnabled(false);
    ui->btnStopThread->setEnabled(true);
}
```

两个线程启动的先后顺序不应调换, 应先启动 threadConsumer, 使其先进入 wait 状态, 后启动 threadProducer, 这样在 threadProducer 里 wakeAll()时 threadConsumer 就可以及时响应, 否则会

丢失第一次掷骰子的数据。

“结束线程”按钮的代码如下：

```
void Dialog::on_btnStopThread_clicked()
{ //结束线程
    threadProducer.stopThread(); //结束线程的 run() 函数执行
    threadProducer.wait();
    threadConsumer.terminate(); //可能处于等待状态，用 terminate 强制结束
    threadConsumer.wait();
    ui->btnStartThread->setEnabled(true);
    ui->btnStopThread->setEnabled(false);
}
```

结束线程时，若按照上面的顺序先结束 threadProducer 线程，则必须使用 terminate() 来强制结束 threadConsumer 线程，因为 threadConsumer 可能还处于条件等待的阻塞状态中，将无法正常运行线程。

### 13.2.5 基于信号量的线程同步

#### 1. 信号量的原理

信号量 (Semaphore) 是另一种限制对共享资源进行访问的线程同步机制，它与互斥量 (Mutex) 相似，但是有区别。一个互斥量只能被锁定一次，而信号量可以多次使用。信号量通常用来保护一定数量的相同的资源，如数据采集时的双缓冲区。

QSemaphore 是实现信号量功能的类，它提供以下几个基本的函数：

- acquire(int n) 尝试获得 n 个资源。如果没有这么多资源，线程将阻塞直到有 n 个资源可用；
- release(int n) 释放 n 个资源，如果信号量的资源已全部可用之后再 release()，就可以创建更多的资源，增加可用资源的个数；
- int available() 返回当前信号量可用的资源个数，这个数永远不可能为负数，如果为 0，就说明当前没有资源可用；
- bool tryAcquire(int n = 1)，尝试获取 n 个资源，不成功时不阻塞线程。

定义 QSemaphore 的实例时，可以传递一个数值作为初始可用的资源个数。

下面的一段示意代码，说明 QSemaphore 的几个函数的作用。

```
QSemaphore WC(5); // WC.available() == 5, 初始资源个数为 5 个
WC.acquire(4);    // WC.available() == 1, 用了 4 个资源，还剩余 1 个可用
WC.release(2);    // WC.available() == 3, 释放了 2 个资源，剩余 3 个可用
WC.acquire(3);    // WC.available() == 0, 又用了 3 个资源，剩余 0 个可用
WC.tryAcquire(1); // 因为 WC.available() == 0, 返回 false,
WC.acquire();     // 因为 WC.available() == 0, 没有资源可用，阻塞
```

为了理解信号量及上面这段代码的意义，可以假想变量 WC 是一个公共卫生间，初始化时定义 WC 有 5 个位置可用。

- WC.acquire(4)，成功进去 4 个人，占用了 4 个位置，还剩余 1 个位置；
- WC.release(2)，出来了 2 个人，剩余 3 个位置可用；
- WC.acquire(3)，又进去 3 个人，剩余 0 个位置可用；



- WC.tryAcquire(1), 有一个人尝试进去, 但是因为没位置了, 他不等待, 走了, tryAcquire() 函数返回 false;
- WC.acquire(), 有一个人必须进去, 但是因为没位置了, 他就一直在外面等着, 直到有其他人出来, 空余出位置来。

互斥量相当于列车上的卫生间, 一次只允许一个人进出, 信号量则是多人公共卫生间, 允许多人进出。n 个资源就是信号量需要保护的共享资源, 至于资源如何分配, 就是内部处理的问题了。

## 2. 双缓冲区数据采集和读取线程类设计

信号量通常用来保护一定数量的相同的资源, 如数据采集时的双缓冲区, 适用于 Producer/Consumer 模型。

在实例 samp13\_5 中, 创建类似于 Producer/Consumer 模型的两个线程类 QThreadDAQ 和 QThreadShow。qmythread.h 文件中这两个类的定义如下:

```
class QThreadDAQ : public Qthread
{
    Q_OBJECT
private:
    bool    m_stop=false; //停止线程
protected:
    void    run() Q_DECL_OVERRIDE;
public:
    QThreadDAQ();
    void    stopThread();
};

class QThreadShow : public Qthread
{
    Q_OBJECT
private:
    bool    m_stop=false; //停止线程
protected:
    void    run() Q_DECL_OVERRIDE;
public:
    QThreadShow();
    void    stopThread();
signals:
    void    newValue(int *data,int count, int seq);
};
```

QThreadDAQ 是数据采集线程, 例如在使用数据采集卡进行连续数据采集时, 需要一个单独的线程将采集卡采集的数据读取到缓冲区内。

QThreadShow 是数据读取线程, 用于读取已存满数据的缓冲区中的数据并传递给主线程显示, 采用信号与槽机制与主线程交互。

QThreadDAQ/QThreadShow 类的定义与使用 QWaitCondition 的实例 samp13\_4 中的 QThreadProducer/QThreadConsumer 类的定义类似, 只是 QThreadShow 的信号 newValue() 采用了指针作为传递参数, 用于一次传递出一个缓冲区的数据。

qmythread.cpp 文件中 QThreadDAQ 和 QThreadShow 的主要功能代码如下:

```
#include    "qmythread.h"
#include    <QSemaphore>
```

```

const int BufferSize = 8;
int buffer1[BufferSize];
int buffer2[BufferSize];

int curBuf=1; //当前正在写入的 Buffer
int bufNo=0; //采集的缓冲区序号
quint8 counter=0; //数据生成器

QSemaphore emptyBufs(2); //信号量, 空的缓冲区个数, 初始资源个数为 2
QSemaphore fullBufs; //满的缓冲区个数, 初始资源为 0

void QThreadDAQ::run()
{
    m_stop=false; //启动线程时令 m_stop=false
    bufNo=0; //缓冲区序号
    curBuf=1; //当前写入使用的缓冲区
    counter=0; //数据生成器
    int n=emptyBufs.available();
    if (n<2) //保证线程启动时 emptyBufs.available==2
        emptyBufs.release(2-n);
    while(!m_stop) //循环主体
    {
        emptyBufs.acquire(); //获取一个空的缓冲区
        for(int i=0; i<BufferSize; i++) //产生一个缓冲区的数据
        {
            if (curBuf==1)
                buffer1[i]=counter; //向缓冲区写入数据
            else
                buffer2[i]=counter;
            counter++; //模拟数据采集卡产生数据
            msleep(50); //每 50ms 产生一个数
        }
        bufNo++; //缓冲区序号
        if (curBuf==1) // 切换当前写入缓冲区
            curBuf=2;
        else
            curBuf=1;
        fullBufs.release(); //有了一个满的缓冲区, available==1
    }
}

void QThreadShow::run()
{
    m_stop=false;
    int n=fullBufs.available();
    if (n>0)
        fullBufs.acquire(n); //将 fullBufs 可用资源个数初始化为 0
    while(!m_stop) //循环主体
    {
        fullBufs.acquire(); //等待有缓冲区满, 当 fullBufs.available==0 阻塞
        int bufferData[BufferSize];
        int seq=bufNo;

        if (curBuf==1) //当前在写入的缓冲区是 1, 那么满的缓冲区是 2
            for (int i=0; i<BufferSize; i++)

```

```

        bufferData[i]=buffer2[i]; //快速拷贝缓冲区数据
    else
        for (int i=0;i<BufferSize;i++)
            bufferData[i]=buffer1[i];

    emptyBufs.release(); //释放一个空缓冲区
    emit    newValueChanged(bufferData,BufferSize,seq); //给主线程传递数据
}
}

```

在共享变量区定义了两个缓冲区 `buffer1` 和 `buffer2`，都是长度为 `BufferSize` 的数组。

变量 `curBuf` 记录当前写入操作的缓冲区编号，其值只能是 1 或 2，表示 `buffer1` 或 `buffer2`，`bufNo` 是累积的缓冲区个数编号，`counter` 是模拟采集数据的变量。

信号量 `emptyBufs` 初始资源个数为 2，表示有 2 个空的缓冲区可用。

信号量 `fullBufs` 初始化资源个数为 0，表示写满数据的缓冲区个数为零。

`QThreadDAQ::run()` 采用双缓冲方式进行模拟数据采集，线程启动时初始化共享变量，特别的是使 `emptyBufs` 的可用资源个数初始化为 2。

在 `while` 循环体里，第一行语句 `emptyBufs.acquire()` 使信号量 `emptyBufs` 获取一个资源，即获取一个空的缓冲区。用于数据缓存的有两个缓冲区，只要有一个空的缓冲区，就可以向这个缓冲区写入数据。

`while` 循环体里的 `for` 循环每隔 50 毫秒使 `counter` 值加 1，然后写入当前正在写入的缓冲区，当前写入哪个缓冲区由 `curBuf` 决定。`counter` 是模拟采集的数据，连续增加可以判断采集的数据是否连续。

完成 `for` 循环后正好写满一个缓冲区，这时改变 `curBuf` 的值，切换用于写入的缓冲区。

写满一个缓冲区之后，使用 `fullBufs.release()` 为信号量 `fullBufs` 释放一个资源，这时 `fullBufs.available==1`，表示有一个缓冲区被写满了。这样，`QThreadShow` 线程里使用 `fullBufs.acquire()` 就可以获得一个资源，可以读取已写满的缓冲区里的数据。

`QThreadShow::run()` 用于监测是否有已经写满数据的缓冲区，只要有缓冲区写满了数据，就立刻读取数据，然后释放这个缓冲区给 `QThreadDAQ` 线程用于写入。

`QThreadShow::run()` 函数的初始化部分使 `fullBufs.available==0`，即线程刚启动时是没有资源的。

在 `while` 循环体里第一行语句就是通过 `fullBufs.acquire()` 以阻塞方式获取一个资源，只有当 `QThreadDAQ` 线程里写满一个缓冲区，执行一次 `fullBufs.release()` 后，`fullBufs.acquire()` 才获得资源并执行后面的代码。后面的代码就立即用临时变量将缓冲区里的数据读取出来，再调用 `emptyBufs.release()` 给信号量 `emptyBufs` 释放一个资源，然后发射信号 `newValue`，由主线程读取数据并显示。

所以，这里使用了双缓冲区、两个信号量实现采集和读取两个线程的协调操作。采集线程里使用 `emptyBufs.acquire()` 获取可以写入的缓冲区。

实际使用数据采集卡进行连续数据采集时，采集线程是不能停顿下来的，也就是说万一读取线程执行较慢，采集线程是不会等待的。所以实际情况下，读取线程的操作应该比采集线程快。

### 3. QThreadDAQ 和 QThreadShow 的使用

设计窗口基于 QDialog 应用程序 samp13\_5，对话框的类定义如下（省略了一些不重要的或与前面实例重复的部分内容）：

```
class Dialog : public QDialog
{
    Q_OBJECT
private:
    QThreadDAQ    threadProducer;
    QThreadShow    threadConsumer;
private slots:
    void    onthreadB_newValue(int *data, int count, int bufNo);
};
```

Dialog 类定义了两个线程的实例，threadProducer 和 threadConsumer。

自定义了一个槽函数 onthreadB\_newValue()，用于与 threadConsumer 的信号关联，在 Dialog 的构造函数里进行了关联。

```
connect(&threadConsumer, SIGNAL(newValue(int*,int,int)),
        this, SLOT(onthreadB_newValue(int*,int,int)));
```

槽函数 onthreadB\_newValue()的功能就是读取一个缓冲区里的数据并显示，其实现代码如下：

```
void Dialog::onthreadB_newValue(int *data, int count, int bufNo)
{ //读取 threadConsumer 传递的缓冲区的数据
    QString str=QString::asprintf("第 %d 个缓冲区: ",bufNo);
    for (int i=0;i<count;i++)
    {
        str=QString::asprintf("%d, ",*data);
        data++;
    }
    str=str+'\n';
    ui->plainTextEdit->appendPlainText(str);
}
```

传递的指针型参数 int \*data 是一个数组指针，count 是缓冲区长度。

“启动线程”和“结束线程”两个按钮的代码如下（省略了按键使能控制的代码）：

```
void Dialog::on_btnStartThread_clicked()
{ //启动线程
    threadConsumer.start();
    threadProducer.start();
}
void Dialog::on_btnStopThread_clicked()
{ //结束线程
    threadConsumer.terminate();
    threadConsumer.wait();
    threadProducer.terminate();
    threadProducer.wait();
}
```

启动线程时，先启动 threadConsumer，再启动 threadProducer，否则可能丢失第 1 个缓冲区的数据。

结束线程时，都采用 terminate()函数强制结束线程，因为两个线程之间有互锁的关系，若不



使用 `terminate()` 强制结束会出现线程无法结束的问题。

程序运行时的界面如图 13-3 所示。



图 13-3 实例 samp13\_5 运行界面

从图 13-3 可以看出, 没有出现丢失缓冲区或数据点的情况, 两个线程之间协调的很好, 将 `QThreadDAQ::run()` 函数中模拟采样率的延时时间调整为 2 毫秒也没问题 (正常设置为 50 毫秒)。

在实际的数据采集中, 要保证不丢失缓冲区或数据点, 数据读取线程的速度必须快过数据写入缓冲区的线程的速度。

Qt 网络模块提供了用于编写 TCP/IP 客户端和服务端程序的各种类，如用于 TCP 通信的 `QTcpSocket` 和 `QTcpServer`，用于 UDP 通信的 `QUdpSocket`，还有用于实现 HTTP、FTP 等普通网络协议的高级类如 `QNetworkRequest`，`QNetworkReply` 和 `QNetworkAccessManager`。Qt 网络模块还提供用于网络代理、网络承载管理的类，提供基于安全套接字层（Secure Sockets Layer，SSL）协议的安全网络通信的类。

本章主要介绍基本的 TCP 和 UDP 网络通信类的使用，基于 HTTP 的网络下载管理的实现。要在程序中使用 Qt 网络模块，需要在项目配置文件中增加一条配置语句：

```
Qt += network
```

## 14.1 主机信息查询

### 14.1.1 QHostInfo 和 QNetworkInterface 类

查询一个主机的 MAC 地址或 IP 地址是网络应用程序中经常用到的功能，Qt 提供了 `QHostInfo` 和 `QNetworkInterface` 类可以用于此类信息的查询。

`QHostInfo` 的静态函数 `localHostName()` 可获取本机的主机名，静态函数 `fromName()` 可以通过主机名获取 IP 地址，静态函数 `lookupHost()` 可以通过一个主机名，以异步方式查找这个主机的 IP 地址。表 14-1 是 `QHostInfo` 类主要的功能函数（省略了函数中的 `const` 关键字）。

表 14-1 QHostInfo 类的主要函数

类别	函数原型	作用
公共函数	<code>QList&lt;QHostAddress&gt; addresses()</code>	返回与 <code>hostName()</code> 主机关联的 IP 地址列表
	<code>HostInfoError error()</code>	如果主机查找失败，返回失败类型
	<code>QString errorString()</code>	如果主机查找失败，返回错误描述字符串
	<code>QString hostName()</code>	返回通过 IP 查找的主机的名称
	<code>int lookupId()</code>	返回本次查找的 ID
静态函数	<code>void abortHostLookup(int id)</code>	中断主机查找
	<code>QHostInfo fromName(QString &amp;name)</code>	返回指定的主机名的 IP 地址
	<code>QString localDomainName()</code>	返回本机 DNS 域名
	<code>QString localHostName()</code>	返回本机主机名
	<code>int lookupHost(QString &amp;name, QObject *receiver, char *member)</code>	以异步方式根据主机名查找主机的 IP 地址，并返回一个表示本次查找的 ID，可用于 <code>abortHostLookup()</code>

QNetworkInterface 可以获得运行应用程序的主机的所有 IP 地址和网络接口列表。静态函数 allInterfaces() 返回主机上所有的网络接口的列表, 一个网络接口可能包括多个的 IP 地址, 每个 IP 地址与掩码或广播地址关联。如果无需知道子网掩码和广播的 IP 地址, 使用静态函数 allAddresses() 可以获得主机上的所有 IP 地址列表。表 14-2 是 QNetworkInterface 类的主要功能函数。

表 14-2 QNetworkInterface 类的主要函数

类别	函数原型	作用
公共函数	QList<QNetworkAddressEntry> addressEntries()	返回该网络接口 (包括子网掩码和广播地址) 的 IP 地址列表
	QString hardwareAddress()	返回该接口的低级硬件地址, 以太网里就是 MAC 地址
	QString humanReadableName()	返回可以读懂的接口名称, 如果名称不确定, 得到的就是 name() 函数的返回值
	bool isValid()	如果接口信息有效就返回 true
	QString name()	返回网络接口名称
静态函数	QList<QHostAddress> allAddresses()	返回主机上所有 IP 地址的列表
	QList<QNetworkInterface> allInterfaces()	返回主机上所有接口的网络列表

为演示这两个类的主要功能, 创建一个窗口基于 QDialog 的应用程序 samp14\_1, 实例运行界面如图 14-1 所示。对话框界面由 UI 设计器设计, 主要代码都是各按钮的 clicked() 信号的槽函数。



图 14-1 实例 samp14\_1 运行界面

## 14.1.2 QHostInfo 的使用

### 1. 显示本机地址信息

图 14-1 窗口上的 “QHostInfo 获取本机主机名和 IP 地址” 按钮的响应代码如下:

```
void Dialog::on_btnGetHostInfo_clicked()
{
    //QHostInfo 获取主机信息
    QString hostName=QHostInfo::localHostName(); //本地主机名
    ui->plainTextEdit->appendPlainText("本机主机名: "+hostName+"\n");
    QHostInfo hostInfo=QHostInfo::fromName(hostName); //本机 IP 地址

    QList<QHostAddress> addList=hostInfo.addresses(); //IP 地址列表
    if (!addList.isEmpty())
        for (int i=0;i<addList.count();i++)
```

```

{
    QHostAddress aHost=addList.at(i); //每一项是一个 QHostAddress
    bool show=ui->chkOnlyIPv4->isChecked(); //只显示 IPv4
    if (show)
        show=(QAbstractSocket::IPv4Protocol==aHost.protocol()); //IPv4 协议
    else
        show=true;
    if (show) {
        ui->plainTextEdit->appendPlainText("协 议: "+ protocolName(aHost.protocol()));
        ui->plainTextEdit->appendPlainText("本机 IP 地址: "+aHost.toString());
        ui->plainTextEdit->appendPlainText("");
    }
}
}

```

这段代码先通过静态函数 `QHostInfo::localHostName()` 获取本机主机名 `hostName`，然后再使用主机名作为参数，用静态函数 `QHostInfo::fromName(hostName)` 获取主机的信息 `hostInfo`。

`hostInfo` 是 `QHostInfo` 类的实例，通过其函数 `addresses()` 获取主机的 IP 地址列表。

```
QList<QHostAddress> addList=hostInfo.addresses();
```

返回的 `addList` 是 `QHostAddress` 类的列表，`QHostAddress` 类提供一个 IP 地址的信息，包括 IPv4 地址和 IPv6 地址。`QHostAddress` 有以下两个主要的函数。

- `protocol()` 返回 `QAbstractSocket::NetworkLayerProtocol` 类型变量，表示当前 IP 地址的协议类型。`QAbstractSocket::NetworkLayerProtocol` 枚举类型的取值见表 14-3。

表 14-3 `QAbstractSocket::NetworkLayerProtocol` 枚举类型取值

枚举值	表示的协议类型
<code>QAbstractSocket::IPv4Protocol</code>	IPv4
<code>QAbstractSocket::IPv6Protocol</code>	IPv6
<code>QAbstractSocket::AnyIPProtocol</code>	IPv4 或 IPv6
<code>QAbstractSocket::UnknownNetworkLayerProtocol</code>	其他类型

- `toString()` 返回 IP 地址的字符串，表示程序中显示了 IP 地址列表中每个 IP 地址的协议类型和 IP 地址字符串，为根据 `protocol()` 返回的 `QAbstractSocket::NetworkLayerProtocol` 枚举值显示协议名称字符串，自定义了一个函数 `protocolName()`，代码如下：

```

QString Dialog::protocolName(QAbstractSocket::NetworkLayerProtocol protocol)
{ //通过协议类型返回协议名称
    switch(protocol)
    {
        case QAbstractSocket::IPv4Protocol:
            return "IPv4 Protocol";
        case QAbstractSocket::IPv6Protocol:
            return "IPv6 Protocol";
        case QAbstractSocket::AnyIPProtocol:
            return "Any IP Protocol";
        default:
            return "Unknown Network Layer Protocol";
    }
}

```



单击“QHostInfo 获取本机主机名和 IP 地址”按钮，如果勾选了“只显示 IPv4 协议地址”复选框，就只显示本机的 IPv4 地址，否则显示所有 IP 地址信息。

## 2. 查找主机地址信息

QHostInfo 的静态函数 lookupHost()可以根据主机名、域名或 IP 地址查找主机的地址信息，lookupHost()函数原型如下：

```
int QHostInfo::lookupHost(const QString &name, QObject *receiver, const char *member)
```

输入参数 name 是表示主机名的字符串，可以是一个主机名、一个域名，或者是一个 IP 地址。

lookupHost()以异步方式查找主机地址，参数 receiver 和 member 指定一个响应槽函数的接收者和槽函数名称。执行 lookupHost()后，程序可能需要花一定时间来查找主机地址，但不会阻塞程序的运行。当查找到主机地址后，通过信号通知设定的槽函数，在槽函数里读取查找的结果。函数返回一个表示查找的 ID。

图 14-1 中的“QHostInfo 查找域名的 IP 地址”按钮的槽函数及 lookupHost()函数关联槽函数代码如下：

```
void Dialog::on_btnLookup_clicked()
{ // 查找主机信息
    QString hostname=ui->editHost->text(); // 主机名
    ui->plainTextEdit->appendPlainText("正在查找查找主机信息: "+hostname);
    QHostInfo::lookupHost(hostname, this, SLOT(lookedUpHostInfo(QHostInfo)));
}

void Dialog::lookedUpHostInfo(const QHostInfo &host)
{ // 查找主机信息的槽函数
    QList<QHostAddress> addList=host.addresses();
    if (!addList.isEmpty())
        for (int i=0; i<addList.count(); i++)
        {
            QHostAddress aHost=addList.at(i);
            bool show=ui->chkOnlyIPv4->isChecked(); // 只显示 IPv4
            if (show)
                show=QAbstractSocket::IPv4Protocol==aHost.protocol();
            else
                show=true;
            if (show) {
                ui->plainTextEdit->appendPlainText("协议: "+ protocolName(aHost.protocol()));
                ui->plainTextEdit->appendPlainText(aHost.toString());
            }
        }
}
```

### 14.1.3 QNetworkInterface 的使用

QNetworkInterface 可以获得应用程序所在主机的所有网络接口，包括其子网掩码和广播地址等。

静态函数 QNetworkInterface::allInterfaces()获取所有网络接口的列表，函数原型为：

```
QList<QNetworkInterface> QNetworkInterface::allInterfaces()
```

其返回结果是一个 QNetworkInterface 类的列表。

界面上“QNetworkInterface::allInterfaces()”按钮的响应代码如下：

```

void Dialog::on_btnALLInterface_clicked()
{
    //QNetworkInterface::allInterfaces() 函数的使用
    QList<QNetworkInterface> list=QNetworkInterface::allInterfaces();
    for(int i=0;i<list.count();i++)
    {
        QNetworkInterface aInterface=list.at(i);
        if (!aInterface.isValid())
            continue;

        ui->plainTextEdit->appendPlainText("设备名称: "+ aInterface.humanReadableName());
        ui->plainTextEdit->appendPlainText("硬件地址: "+ aInterface.hardwareAddress());
        QList<QNetworkAddressEntry> entryList=aInterface.addressEntries();
        for(int j=0;j<entryList.count();j++)
        {
            QNetworkAddressEntry aEntry=entryList.at(j);
            ui->plainTextEdit->appendPlainText(" IP 地址: "+ aEntry.ip().toString());
            ui->plainTextEdit->appendPlainText(" 子网掩码: "+ aEntry.netmask().toString());
            ui->plainTextEdit->appendPlainText(" 广播地址: "+ aEntry.broadcast().
toString()+"\n");
        }
        ui->plainTextEdit->appendPlainText("\n");
    }
}

```

通过 `QNetworkInterface::allInterfaces()` 获取网络接口列表 `list` 之后, 显示每个接口的 `humanReadableName()` 和 `hardwareAddress()`。每个接口又有一个 `QNetworkAddressEntry` 类型的地址列表, 通过 `addressEntries()` 获得这个列表。

`QNetworkAddressEntry` 包含了一个网络接口的 IP 地址、子网掩码和广播地址, 分别用 `ip()`、`netmask()` 和 `broadcast()` 函数返回。

`QNetworkInterface::allInterfaces()` 返回的网络接口的信息很多, 如果无需知道子网掩码和广播地址等信息, 可以使用 `QNetworkInterface::allAddresses()` 只获取 IP 地址。

界面上 “`QNetworkInterface::allAddresses()`” 按钮的响应代码如下:

```

void Dialog::on_btnDetail_clicked()
{
    //QNetworkInterface::allAddresses() 的使用
    QList<QHostAddress> addList=QNetworkInterface::allAddresses();
    if (!addList.isEmpty())
        for (int i=0;i<addList.count();i++)
        {
            QHostAddress aHost=addList.at(i);
            bool show=ui->chkOnlyIPv4->isChecked(); //只显示 IPv4
            if (show)
                show=QAbstractSocket::IPv4Protocol==aHost.protocol();
            else
                show=true;
            if (show) {
                ui->plainTextEdit->appendPlainText("协 议: "+ protocolName(aHost.protocol()));
                ui->plainTextEdit->appendPlainText("IP 地址: "+aHost.toString());
                ui->plainTextEdit->appendPlainText("");
            }
        }
}

```

QNetworkInterface::allAddresses()的功能与 QHostInfo::addresses()函数功能相似，都是返回一个 QHostAddress 的列表。只是 QNetworkInterface 会返回更多地址，包括表示本机的 127.0.0.1，而 QHostInfo 不会返回这个 IP 地址。

## 14.2 TCP 通信

### 14.2.1 TCP 通信概述

TCP(Transmission Control Protocol)是一种被大多数 Internet 网络协议（如 HTTP 和 FTP）用于数据传输的低级网络协议，它是可靠的、面向流、面向连接的传输协议，特别适合用于连续数据传输。

TCP 通信必须先建立 TCP 连接，通信端分为客户端和服务端（如图 14-2 所示）。Qt 提供 QTcpSocket 类和 QTcpServer 类用于建立 TCP 通信应用程序。服务器端程序必须使用 QTcpServer 用于端口监听，建立服务器；QTcpSocket 用于建立连接后使用套接字(Socket)进行通信。

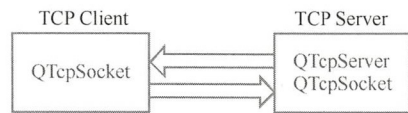


图 14-2 客户端与服务端 TCP 通信示意图

QTcpServer 是从 QObject 继承的类，它主要用于服务器端建立网络监听，创建网络 Socket 连接。QTcpServer 类的主要接口函数见表 14-4（省略了函数中的 const 关键字，省略了缺省参数）。

表 14-4 QTcpServer 类的主要接口函数

类型	函数	功能
公共函数	void close()	关闭服务器，停止网络监听
	bool listen()	在给定的 IP 地址和端口上开始监听，若成功就返回 true
	bool isListening()	返回 true 表示服务器处于监听状态
	QTcpSocket * nextPendingConnection()	返回下一个等待接入的连接
	QHostAddress serverAddress()	如果服务器处于监听状态，返回服务器地址
	quint16 serverPort()	如果服务器处于监听状态，返回服务器监听端口
	bool waitForNewConnection()	以阻塞方式等待新的连接
信号	void acceptError( QAbstractSocket::SocketError socketError )	当接受一个新的连接发生错误时发射此信号，参数 socketError 描述了错误信息
	void newConnection()	当有新的连接时发射此信号
保护函数	void incomingConnection(qintptr socketDescriptor)	当有一个新的连接可用时，QTcpServer 内部调用此函数，创建一个 QTcpSocket 对象，添加到内部可用新连接列表，然后发射 newConnection()信号。用户若从 QTcpServer 继承定义类，可以重定义此函数，但必须调用 addPendingConnection()
	void addPendingConnection(QTcpSocket *socket)	由 incomingConnection()调用，将创建的 QTcpSocket 添加到内部新可用连接列表

服务器端程序首先需要用 QTcpServer::listen()开始服务器端监听，可以指定监听的 IP 地址和端口，一般一个服务程序只监听某个端口的网络连接。

当有新的客户端接入时，QTcpServer 内部的 incomingConnection()函数会创建一个与客户端连接的 QTcpSocket 对象，然后发射信号 newConnection()。在 newConnection()信号的槽函数中，可

以用 `nextPendingConnection()` 接受客户端的连接，然后使用 `QTcpSocket` 与客户端通信。

所以在客户端与服务器建立 TCP 连接后，具体的数据通信是通过 `QTcpSocket` 完成的。`QTcpSocket` 类提供了 TCP 协议的接口，可以用 `QTcpSocket` 类实现标准的网络通信协议如 POP3、SMTP 和 NNTP，也可以设计自定义协议。

`QTcpSocket` 是从 `QIODevice` 间接继承的类，所以具有流读写的功能。`QTcpSocket` 和下一节要讲到的 `QUdpSocket` 的类继承关系如图 14-3 所示。

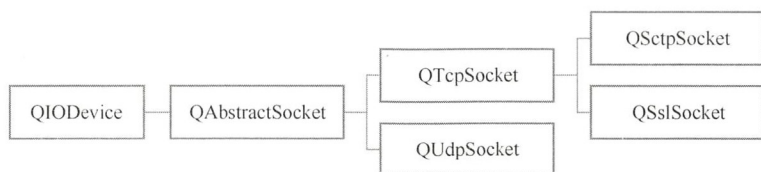


图 14-3 `QTcpSocket` 和 `QUdpSocket` 的类继承关系

`QTcpSocket` 类除了构造函数和析构函数，其他函数都是从 `QAbstractSocket` 继承或重定义的。`QAbstractSocket` 用于 TCP 通信的主要接口函数见表 14-5（省略了函数中的 `const` 关键字，省略了缺省参数）。

表 14-5 `QAbstractSocket` 类的主要接口函数

类型	函数	功能
公共函数	<code>void connectToHost(QHostAddress &amp;address, quint16 port,)</code>	以异步方式连接到指定 IP 地址和端口的 TCP 服务器，连接成功后会发射 <code>connected()</code> 信号
	<code>void disconnectFromHost()</code>	断开 socket，关闭成功后发射 <code>disconnected()</code> 信号
	<code>bool waitForConnected()</code>	等待直到建立 socket 连接
	<code>bool waitForDisconnected()</code>	等待直到断开 socket 连接
	<code>QHostAddress localAddress()</code>	返回本 socket 的地址
	<code>quint16 localPort()</code>	返回本 socket 的端口
	<code>QHostAddress peerAddress()</code>	在已连接状态下，返回对方 socket 的地址
	<code>QString peerName()</code>	返回 <code>connectToHost()</code> 连接到的对方的主机名
	<code>quint16 peerPort()</code>	在已连接状态下，返回对方 socket 的端口
	<code>qint64 readBufferSize()</code>	返回内部读取缓冲区的大小，该大小决定了 <code>read()</code> 和 <code>readAll()</code> 函数能读出的数据的大小
	<code>void setReadBufferSize(qint64 size)</code>	设置内部读取缓冲区大小
	<code>qint64 bytesAvailable()</code>	返回需要读取的缓冲区的数据的字节数
	<code>bool canReadLine()</code>	如果有行数据要从 socket 缓冲区读取，就返回 <code>true</code>
	<code>SocketState state()</code>	返回 socket 当前的状态
信号	<code>void connected()</code>	<code>connectToHost()</code> 成功连接到服务器后发射此信号
	<code>void disconnected()</code>	当 socket 断开连接后发射此信号
	<code>void error(QAbstractSocket::SocketError socketError)</code>	当 socket 发生错误时发射此信号
	<code>void hostFound()</code>	调用 <code>connectToHost()</code> 找到主机后发射此信号
	<code>void stateChanged(QAbstractSocket::SocketState socketState)</code>	当 socket 的状态变化时发射此信号，参数 <code>socketState</code> 表示了 socket 当前的状态
	<code>void readyRead()</code>	当缓冲区有新数据需要读取时发射此信号，在此信号的槽函数里读取缓冲区的数据

TCP 客户端使用 `QTcpSocket` 与 TCP 服务器建立连接并通信。



客户端的 `QTcpSocket` 实例首先通过 `connectToHost()` 尝试连接到服务器, 需要指定服务器的 IP 地址和端口。`connectToHost()` 是异步方式连接服务器, 不会阻塞程序运行, 连接后发射 `connected()` 信号。

如果需要使用阻塞方式连接服务器, 则使用 `waitForConnected()` 函数阻塞程序运行, 直到连接成功或失败。例如:

```
socket->connectToHost("192.168.1.100", 1340);
if (socket->waitForConnected(1000))
    qDebug("Connected!");
```

与服务器端建立 `socket` 连接后, 就可以向缓冲区写数据或从接收缓冲区读取数据, 实现数据的通信。当缓冲区有新数据进入时, 会发射 `readyRead()` 信号, 一般在此信号的槽函数里读取缓冲区数据。

`QTcpSocket` 是从 `QIODevice` 间接继承的, 所以可以使用流数据读写功能。一个 `QTcpSocket` 实例既可以接收数据也可以发送数据, 且接收与发射是异步工作的, 有各自的缓冲区。

作为演示 TCP 通信的实例, 创建了一个 `TCPClient` 程序和一个 `TCPServer` 程序, 两个程序运行时界面如图 14-4 和图 14-5 所示。



图 14-4 TCPServer 程序

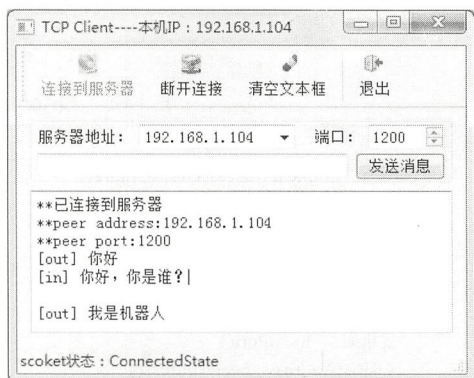


图 14-5 TCPClient 程序

TCPServer 程序具有如下的功能:

- 根据指定 IP 地址 (本机地址) 和端口打开网络监听, 有客户端连接时创建 `socket` 连接;
- 采用基于行的数据通信协议, 可以接收客户端发来的消息, 也可以向客户端发送消息;
- 在状态栏显示服务器监听状态和 `socket` 的状态。

TCPClient 程序具有如下的功能:

- 通过 IP 地址和端口号连接到服务器;
- 采用基于行的数据通信协议, 与服务器端收发消息;
- 处理 `QTcpSocket` 的 `StateChange()` 信号, 在状态栏显示 `socket` 的状态。

## 14.2.2 TCP 服务器端程序设计

### 1. 主窗口定义与构造函数

TCPServer 是一个窗口基于 `QMainWindow` 的应用程序, 界面由 UI 设计器设计, `MainWindow`

类的定义如下（忽略了 UI 设计器自动生成的 actions 和按钮的槽函数）：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QLabel *LabListen; //状态栏标签
    QLabel *LabSocketState; //状态栏标签
    QTcpServer *tcpServer; //TCP 服务器
    QTcpSocket *tcpSocket; //TCP 通信的 Socket
    QString getLocalIP(); //获取本机 IP 地址
protected:
    void closeEvent(QCloseEvent *event);
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    //自定义槽函数
    void onNewConnection(); //QTcpServer 的 newConnection() 信号
    void onSocketStateChange(QAbstractSocket::SocketState socketState);
    void onClientConnected(); //Client Socket connected
    void onClientDisconnected(); //Client Socket disconnected
    void onSocketReadyRead(); //读取 socket 传入的数据
private:
    Ui::MainWindow *ui;
};
```

MainWindow 中定义了私有变量 tcpServer 用于建立 TCP 服务器，定义了 tcpSocket 用于与客户端进行 socket 连接和通信。

定义了几个槽函数，用于与 QTcpServer 和 QTcpSocket 的相关信号连接，实现相应的处理。

MainWindow 构造函数代码如下：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setUpUi(this);
    LabListen=new QLabel("监听状态:");
    LabListen->setMinimumWidth(150);
    ui->statusBar->addWidget(LabListen);

    LabSocketState=new QLabel("Socket 状态: ");
    LabSocketState->setMinimumWidth(200);
    ui->statusBar->addWidget(LabSocketState);

    QString localIP=getLocalIP(); //本机 IP
    this->setWindowTitle(this->windowTitle()+"----本机 IP: "+localIP);
    ui->comboIP->addItem(localIP);
    tcpServer=new QTcpServer(this);
    connect(tcpServer, SIGNAL(newConnection()), this, SLOT(onNewConnection()));
}

QString MainWindow::getLocalIP()
{ //获取本机 IPv4 地址
    QString hostName=QHostInfo::localHostName(); //本地主机名
    QHostInfo hostInfo=QHostInfo::fromName(hostName);
```

```

QString localIP="";
QList<QHostAddress> addList=hostInfo.addresses();

if (!addList.isEmpty())
for (int i=0;i<addList.count();i++)
{
    QHostAddress aHost=addList.at(i);
    if (QAbstractSocket::IPv4Protocol==aHost.protocol())
    { localIP=aHost.toString(); break; }
}
return localIP;
}

```

MainWindow 的构造函数创建状态栏上的标签用于信息显示,调用自定义函数 getLocalIP()获取本机 IP 地址,并显示到标题栏上。创建 QTcpServer 实例 tcpServer,并将其 newConnection()信号与 onNewConnection()槽函数关联。

## 2. 网络监听与 socket 连接的建立

作为 TCP 服务器,QTcpServer 类需要调用 listen()在本机某个 IP 地址和端口上开始 TCP 监听,以等待 TCP 客户端的接入。单击主窗口上“开始监听”按钮可以开始网络监听,其代码如下:

```

void MainWindow::on_actStart_triggered()
{
    //开始监听
    QString IP=ui->comboIP->currentText(); //IP 地址
    quint16 port=ui->spinPort->value(); //端口
    QHostAddress addr(IP);
    tcpServer->listen(addr,port); //开始监听
    ui->plainTextEdit->appendPlainText("**开始监听...");
    ui->plainTextEdit->appendPlainText("**服务器地址: "
        +tcpServer->serverAddress().toString());
    ui->plainTextEdit->appendPlainText("**服务器端口: "
        +QString::number(tcpServer->serverPort()));
    ui->actStart->setEnabled(false);
    ui->actStop->setEnabled(true);
    LabListen->setText("监听状态: 正在监听");
}

```

程序读取窗口上设置的监听地址和监听端口,然后调用 QTcpServer 的 listen()函数开始监听。TCP 服务器在本机上监听,所以 IP 地址可以是表示本机的“127.0.0.1”,或是本机的实际 IP,亦或是常量 QHostAddress::LocalHost,即在本机上监听某个端口也可以写成:

```
tcpServer->listen(QHostAddress::LocalHost,port);
```

tcpServer 开始监听后,TCPCClient 就可以通过 IP 地址和端口连接到此服务器。当有客户端接入时,tcpServer 会发射 newConnection()信号,此信号关联的槽函数 onNewConnection()的代码如下:

```

void MainWindow::onNewConnection()
{
    tcpSocket = tcpServer->nextPendingConnection(); //获取 socket
    connect(tcpSocket, SIGNAL(connected()), this, SLOT(onClientConnected()));
    onClientConnected();
    connect(tcpSocket, SIGNAL(disconnected()),
        this, SLOT(onClientDisconnected()));
    connect(tcpSocket, SIGNAL(stateChanged(QAbstractSocket::SocketState)),

```

```

        this, SLOT(onSocketStateChange(QAbstractSocket::SocketState)));
onSocketStateChange(tcpSocket->state());
connect(tcpSocket, SIGNAL(readyRead()), this, SLOT(onSocketReadyRead()));
}

```

程序首先通过 `nextPendingConnection()` 函数获取与接入连接进行通信的 `QTcpSocket` 对象实例 `tcpSocket`，然后将 `tcpSocket` 的几个信号与相应的槽函数连接起来。`QTcpSocket` 的这几个信号的作用是：

- `connected()` 信号，客户端 socket 连接建立时发射此信号；
- `disconnected()` 信号，客户端 socket 连接断开时发射此信号；
- `stateChanged()`，本程序的 socket 状态变化时发射此信号；
- `readyRead()`，本程序的 socket 的读取缓冲区有新数据时发射此信号。

涉及状态变化的几个信号的槽函数代码如下：

```

void MainWindow::onClientConnected()
{
    //客户端接入时
    ui->plainTextEdit->appendPlainText("**client socket connected");
    ui->plainTextEdit->appendPlainText("**peer address:" +
        tcpSocket->peerAddress().toString());
    ui->plainTextEdit->appendPlainText("**peer port:" +
        QString::number(tcpSocket->peerPort()));
}

void MainWindow::onClientDisconnected()
{
    //客户端断开连接时
    ui->plainTextEdit->appendPlainText("**client socket disconnected");
    tcpSocket->deleteLater();
}

void MainWindow::onSocketStateChange(QAbstractSocket::SocketState socketState)
{
    //socket 状态变化时
    switch(socketState)
    {
        case QAbstractSocket::UnconnectedState:
            LabSocketState->setText("socket 状态: UnconnectedState"); break;
        case QAbstractSocket::HostLookupState:
            LabSocketState->setText("socket 状态: HostLookupState"); break;
        case QAbstractSocket::ConnectingState:
            LabSocketState->setText("socket 状态: ConnectingState"); break;
        case QAbstractSocket::ConnectedState:
            LabSocketState->setText("socket 状态: ConnectedState"); break;
        case QAbstractSocket::BoundState:
            LabSocketState->setText("socket 状态: BoundState"); break;
        case QAbstractSocket::ClosingState:
            LabSocketState->setText("socket 状态: ClosingState"); break;
        case QAbstractSocket::ListeningState:
            LabSocketState->setText("socket 状态: ListeningState");
    }
}

```

TCP 服务器停止监听，只需调用 `QTcpServer` 的 `close()` 函数即可。窗口上的“停止监听”响应代码如下：

```

void MainWindow::on_actStop_triggered()
{
    //停止监听
    if (tcpServer->isListening()) //tcpServer 正在监听

```



```

{
    tcpServer->close(); //停止监听
    ui->actStart->setEnabled(true);
    ui->actStop->setEnabled(false);
    LabListen->setText("监听状态: 已停止监听");
}
}

```

### 3. 与 TCPCClient 的数据通信

TCP 服务器端和客户端之间通过 QTcpSocket 通信时, 需要规定两者之间的通信协议, 即传输的数据内容如何解析。QTcpSocket 间接继承于 QIODevice, 所以支持流读写功能。

Socket 之间的数据通信协议一般有两种方式, 基于行的或基于数据块的。

基于行的数据通信协议一般用于纯文本数据的通信, 每一行数据以一个换行符结束。canReadLine()函数判断是否有新的一行数据需要读取, 再用 readLine()函数读取一行数据, 例如:

```

while(tcpClient->canReadLine())
    ui->plainTextEdit->appendPlainText("[in] "+tcpClient->readLine());

```

基于块的数据通信协议用于一般的二进制数据的传输, 需要自定义具体的格式。

实例程序 TCPServer 和 TCPCClient 只是进行字符串的信息传输, 类似于一个简单的聊天程序, 程序采用基于行的数据通信协议。

单击窗口上的“发送消息”, 将文本框里的字符串发送给客户端, 其实现代码如下:

```

void MainWindow::on_btnSend_clicked()
{ //发送一行字符串, 以换行符结束
    QString msg=ui->editMsg->text();
    ui->plainTextEdit->appendPlainText("[out] "+msg);
    ui->editMsg->clear();
    ui->editMsg->setFocus();

    QByteArray str=msg.toUtf8();
    str.append('\n');//添加一个换行符
    tcpSocket->write(str);
}

```

从上面的代码中可以看到, 读取文本框中的字符串到 msg 后, 先将其转换为 QByteArray 类型字节数组 str, 然后在 str 最后面添加一个换行符, 用 QIODevice 的 write()函数写入缓冲区, 这样就向客户端发送一行文字。

QTcpSocket 接收到数据后, 会发射 readyRead()信号, 在 onNewConnection()槽函数中已经建立了这个信号与槽函数 onSocketReadyRead()的连接。

槽函数 onSocketReadyRead()实现缓冲区数据的读取, 其代码如下:

```

void MainWindow::onSocketReadyRead()
{ //读取缓冲区行文本
    while(tcpSocket->canReadLine())
        ui->plainTextEdit->appendPlainText("[in] "+tcpSocket->readLine());
}

```

这样, TCPServer 就可以与 TCPCClient 之间进行双向通信了, 且这个连接将一直存在, 直到某一方的 QTcpSocket 对象调用 disconnectFromHost()函数断开 socket 连接。

### 14.2.3 TCP 客户端程序设计

#### 1. 主窗口定义与构造函数

客户端程序 TCPClient 只需要使用一个 QTcpSocket 对象，就可以和服务器端程序 TCPServer 进行通信。

TCPClient 也是一个窗口基于 QMainWindow 的应用程序，其主窗口的定义如下：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QTcpSocket *tcpClient; //socket
    QLabel *LabSocketState; //状态栏显示标签
    QString getLocalIP(); //获取本机 IP 地址
protected:
    void closeEvent(QCloseEvent *event);
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    //自定义槽函数
    void onConnected();
    void onDisconnected();
    void onSocketStateChange(QAbstractSocket::SocketState socketState);
    void onSocketReadyRead(); //读取 socket 传入的数据
private:
    Ui::MainWindow *ui;
};
```

这里只定义了一个用于 socket 连接和通信的 QTcpSocket 变量 tcpClient，自定义了几个槽函数，用于与 tcpClient 的相关信号关联。

下面是 MainWindow 的构造函数，主要功能是创建 tcpClient，并建立信号与槽函数的关联。

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    tcpClient=new QTcpSocket(this); //创建 socket 变量
    LabSocketState=new QLabel("Socket 状态: "); //状态栏标签
    LabSocketState->setMinimumWidth(250);
    ui->statusBar->addWidget(LabSocketState);
    QString localIP=getLocalIP(); //本机 IP
    this->setWindowTitle(this->windowTitle()+"----本机 IP: "+localIP);
    ui->comboServer->addItem(localIP);

    connect(tcpClient,SIGNAL(connected()),this,SLOT(onConnected()));
    connect(tcpClient,SIGNAL(disconnected()),this,SLOT(onDisconnected()));
    connect(tcpClient,SIGNAL(stateChanged(QAbstractSocket::SocketState)),
        this,SLOT(onSocketStateChange(QAbstractSocket::SocketState)));
    connect(tcpClient,SIGNAL(readyRead()),
        this,SLOT(onSocketReadyRead()));
}
```

## 2. 与服务器端建立 socket 连接

在窗口上设置服务器 IP 地址和端口后, 调用 QTcpSocket 的函数 connectToHost() 连接到服务器, 也可以使用 disconnectFromHost() 函数断开与服务器的连接。

下面是两个按钮的响应代码, 以及两个相关槽函数的代码:

```
void MainWindow::on_actConnect_triggered()
{
    // “连接到服务器” 按钮
    QString    addr=ui->comboServer->currentText();
    quint16    port=ui->spinPort->value();
    tcpClient->connectToHost(addr,port);
}
void MainWindow::on_actDisconnect_triggered()
{
    // “断开连接” 按钮
    if (tcpClient->state()==QAbstractSocket::ConnectedState)
        tcpClient->disconnectFromHost();
}
void MainWindow::onConnected()
{
    // connected() 信号槽函数
    ui->plainTextEdit->appendPlainText("***已连接到服务器");
    ui->plainTextEdit->appendPlainText("**peer address:"+
        tcpClient->peerAddress().toString());
    ui->plainTextEdit->appendPlainText("**peer port:"+
        QString::number(tcpClient->peerPort()));
    ui->actConnect->setEnabled(false);
    ui->actDisconnect->setEnabled(true);
}
void MainWindow::onDisconnected()
{
    // disconnected() 信号槽函数
    ui->plainTextEdit->appendPlainText("***已断开与服务器的连接");
    ui->actConnect->setEnabled(true);
    ui->actDisconnect->setEnabled(false);
}
}
```

槽函数 onSocketStateChange() 的功能和代码与 TCPServer 中的完全一样, 这里不再赘述。

## 3. 与 TCPServer 的数据收发

TCPClient 与 TCPServer 之间采用基于行的数据通信协议。单击“发送消息”按钮将发送一行字符串。在 readyRead() 信号的槽函数里读取行字符串, 其相关代码如下:

```
void MainWindow::on_btnSend_clicked()
{
    // 发送数据
    QString    msg=ui->editMsg->text();
    ui->plainTextEdit->appendPlainText("[out] "+msg);
    ui->editMsg->clear();
    ui->editMsg->setFocus();
    QByteArray str=msg.toUtf8();
    str.append('\n');
    tcpClient->write(str);
}
void MainWindow::onSocketReadyRead()
{
    // readyRead() 信号槽函数
    while(tcpClient->canReadLine())
        ui->plainTextEdit->appendPlainText("[in] "+tcpClient->readLine());
}
}
```

实例 TCPServer 和 TCPClient 只是简单演示了 TCP 通信的基本原理, TCPServer 只允许一个 TCPClient 客户端接入。而一般的 TCP 服务器程序允许多个客户端接入, 为了使每个 socket 连接独立通信互不影响, 一般采用多线程, 即为一个 socket 连接创建一个线程。

实例 TCPServer 和 TCPClient 之间的数据通信采用基于行的通信协议, 只能传输字符串数据。QTcpSocket 间接继承于 QIODevice, 可以使用数据流的方式传输二进制数据流, 例如传输图片、任意格式文件等, 但是这涉及到服务器端和客户端之间通信协议的定义, 本书不具体介绍了。

## 14.3 QUdpSocket 实现 UDP 通信

### 14.3.1 UDP 通信概述

UDP (User Datagram Protocol, 用户数据报协议) 是轻量的、不可靠的、面向数据报 (datagram)、无连接的协议, 它可以用于对可靠性要求不高的场合。与 TCP 通信不同, 两个程序之间进行 UDP 通信无需预先建立持久的 socket 连接, UDP 每次发送数据报都需要指定目标地址和端口 (如图 14-6 所示)。

QUdpSocket 类用于实现 UDP 通信, 它从 QAbstractSocket 类继承, 因而与 QTcpSocket 共享大部分的接口函数。主要区别是 QUdpSocket 以数据报传输数据, 而不是以连续的数据流。发送数据报使用函数 QUdpSocket::writeDatagram(), 数据报的长度一般少于 512 字节, 每个数据报包含发送者和接收者的 IP 地址和端口等信息。

要进行 UDP 数据接收, 要用 QUdpSocket::bind() 函数先绑定一个端口, 用于接收传入的数据报。当有数据报传入时会发射 readyRead() 信号, 使用 readDatagram() 函数来读取接收到的数据报。

UDP 消息传送有单播、广播、组播三种模式, 其示意图如图 14-7 所示。

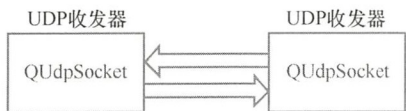


图 14-6 UDP 收发器之间通信示意图

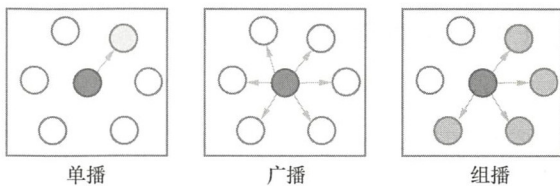


图 14-7 UDP 客户端之间通信的三种模式

- 单播 (unicast) 模式: 一个 UDP 客户端发出的数据报只发送到另一个指定地址和端口的 UDP 客户端, 是一对一的数据传输。
- 广播 (broadcast) 模式: 一个 UDP 客户端发出的数据报, 在同一网络范围内其他所有的 UDP 客户端都可以收到。QUdpSocket 支持 IPv4 广播。广播经常用于实现网络发现的协议。要获取广播数据只需在数据报中指定接收端地址为 QHostAddress::Broadcast, 一般的广播地址是 255.255.255.255。
- 组播 (multicast) 模式: 也称为多播。UDP 客户端加入到另一个组播 IP 地址指定的多播组, 成员向组播地址发送的数据报组内成员都可以接收到, 类似于 QQ 群的功能。QUdpSocket::joinMulticastGroup() 函数实现加入多播组的功能, 加入多播组后, UDP 数据



的收发与正常的 UDP 数据收发方法一样。

使用广播和多播模式，UDP 可以实现一些比较灵活的通信功能，而 TCP 通信只有单播模式，没有广播和多播模式。所以，UDP 通信虽然不能保证数据传输的准确性，但是具有灵活性，一般的即时通信软件都是基于 UDP 通信的。

QUdpSocket 类从 QAbstractSocket 继承而来，但是又定义了较多新的功能函数用于实现 UDP 特有的一些功能，如数据报读写和多播通信功能。QUdpSocket 没有定义新的信号。QUdpSocket 的主要功能函数见表 14-6（包括从 QAbstractSocket 继承的函数，省略了函数中的 const 关键字，省略了缺省参数）。

表 14-6 QUdpSocket 类的主要接口函数

函数	功能
bool bind(quint16 port = 0)	为 UDP 通信绑定一个端口
quint64 writeDatagram(QByteArray &datagram, QHostAddress &host, quint16 port)	向目标地址和端口的 UDP 客户端发送数据报，返回成功发送的字节数
bool hasPendingDatagrams()	当至少有一个数据报需要读取时，返回 true
quint64 pendingDatagramSize()	返回第一个待读取的数据报的大小
quint64 readDatagram(char *data, quint64 maxSize)	读取一个数据报，返回成功读取的数据报的字节数
bool joinMulticastGroup(QHostAddress &groupAddress)	加入一个多播组
bool leaveMulticastGroup(QHostAddress &groupAddress)	离开一个多播组

在单播、广播和多播模式下，UDP 程序都是对等的，不像 TCP 通信那样分为客户端和服务端。多播和广播的实现方式基本相同，只是数据报的目标 IP 地址设置不同，多播模式需要加入多播组，实现方式有较大差异。

为分别演示这三种 UDP 通信模式，本节设计了两个实例。Samp14\_3 实例演示 UDP 单播和广播通信，Samp14\_4 实例演示 UDP 组播通信。

### 14.3.2 UDP 单播和广播

#### 1. UDP 通信实例程序功能

实例程序 samp14\_3 实现 UDP 单播和广播，其主窗口是继承自 QMainWindow 的类，界面用 UI 设计器设计。程序可以进行 UDP 数据报的发送和接收，samp14\_3 的两个运行实例之间可以进行 UDP 通信，这两个实例可以运行在同一台计算机上，也可以运行在不同的计算机上。图 14-8 和图 14-9 是 samp14\_3 两个实例在一台计算机上运行时通信的界面。

在同一台计算机上运行时，两个运行实例需要绑定不同的端口，例如实例 A 绑定端口 1200，实例 B 绑定端口 3355。实例 A 向实例 B 发送数据报时，需要指定实例 B 所在主机的 IP 地址、绑定端口作为目标地址和目标端口，这样实例 B 才能接收到数据报。

如果两个实例在不同计算机上运行，则可以使用相同的端口，因为 IP 地址不同了，不会导致绑定时发生冲突。一般的 UDP 通信程序都是在不同的计算机上运行的，约定一个固定的端口作为通信端口。

#### 2. 主窗口类定义和构造函数

主窗口是基于 QMainWindow 的类 MainWindow，界面采用 UI 设计器设计。MainWindow 类的定义如下（省略了 UI 设计器为 actions 和按钮生成的槽函数声明）：

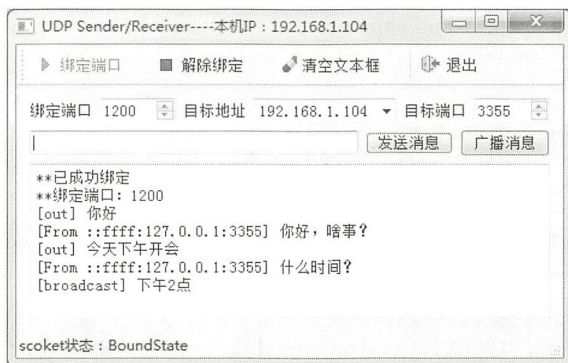


图 14-8 samp14\_3 运行实例 A (绑定端口 1200)



图 14-9 samp14\_3 运行实例 B (绑定端口 3355)

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QLabel *LabSocketState; //socket 状态显示标签
    QUdpSocket *udpSocket;
    QString getLocalIP(); //获取本机 IP 地址
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    //自定义槽函数
    void onSocketStateChange(QAbstractSocket::SocketState socketState);
    void onSocketReadyRead(); //读取 socket 传入的数据
private:
    Ui::MainWindow *ui;
};
```

QUdpSocket 类型的私有变量 udpSocket 是用于 UDP 通信的 socket。

定义了两个自定义槽函数，onSocketStateChange()与 udpSocket 的 stateChange()信号关联，用于显示 udpSocket 当前的状态；onSocketReadyRead()信号与 udpSocket 的 readyRead()信号关联，用于读取缓冲区的数据报。

MainWindow 的构造函数主要完成 udpSocket 的创建、信号与槽函数的关联，代码如下：

```

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setUpUi(this);
    LabSocketState=new QLabel("Socket 状态: ");
    LabSocketState->setMinimumWidth(200);
    ui->statusBar->addWidget(LabSocketState);
    QString localIP=getLocalIP();//本机 IP
    this->setWindowTitle(this->windowTitle()+"----本机 IP: "+localIP);
    ui->comboTargetIP->addItem(localIP);

    udpSocket=new QUdpSocket(this);
    connect(udpSocket,SIGNAL(stateChanged(QAbstractSocket::SocketState)),
        this,SLOT(onSocketStateChange(QAbstractSocket::SocketState)));
    onSocketStateChange(udpSocket->state());
    connect(udpSocket,SIGNAL(readyRead()),this,SLOT(onSocketReadyRead()));
}

```

槽函数 `onSocketStateChange()` 的功能与 14.2 节 TCP 通信程序里的完全一样，不再显示其具体代码。

### 3. UDP 通信的实现

要实现 UDP 数据的接收，必须先用 `QUdpSocket::bind()` 函数绑定一个端口，用于监听传入的数据报，解除绑定则使用 `abort()` 函数。程序主窗口上的“绑定端口”和“解除绑定”按钮的响应代码如下：

```

void MainWindow::on_actStart_triggered()
{ // 绑定端口
    quint16 port=ui->spinBindPort->value(); // 本机 UDP 端口
    if (udpSocket->bind(port)) // 绑定端口成功
    {
        ui->plainTextEdit->appendPlainText("***已成功绑定");
        ui->plainTextEdit->appendPlainText("***绑定端口: "
            +QString::number(udpSocket->localPort()));
        ui->actStart->setEnabled(false);
        ui->actStop->setEnabled(true);
    }
    else
        ui->plainTextEdit->appendPlainText("***绑定失败");
}

void MainWindow::on_actStop_triggered()
{ // 解除绑定
    udpSocket->abort(); // 解除绑定
    ui->actStart->setEnabled(true);
    ui->actStop->setEnabled(false);
    ui->plainTextEdit->appendPlainText("***已解除绑定");
}

```

绑定端口后，socket 的状态变为已绑定状态“BoundState”，解除绑定后状态变为未连接状态“UnconnectedState”。

发送点对点消息和广播消息都使用 `QUdpSocket::writeDatagram()` 函数，窗口上“发送消息”和“广播消息”两个按钮的代码如下：

```

void MainWindow::on_btnSend_clicked()
{ // 发送消息 按钮
    QString targetIP=ui->comboTargetIP->currentText(); // 目标 IP

```

```

QHostAddress targetAddr(targetIP);
quint16 targetPort=ui->spinTargetPort->value(); //目标 port
QString msg=ui->editMsg->text(); //发送的消息内容
QByteArray str=msg.toUtf8();
udpSocket->writeDatagram(str,targetAddr,targetPort); //发出数据报
ui->plainTextEdit->appendPlainText("[out] "+msg);
ui->editMsg->clear();
ui->editMsg->setFocus();
}

void MainWindow::on_btnBroadcast_clicked()
{ //广播消息 按钮
    quint16 targetPort=ui->spinTargetPort->value(); //目标端口
    QString msg=ui->editMsg->text();
    QByteArray str=msg.toUtf8();
    udpSocket->writeDatagram(str,QHostAddress::Broadcast,targetPort);
    ui->plainTextEdit->appendPlainText("[broadcast] "+msg);
    ui->editMsg->clear();
    ui->editMsg->setFocus();
}

```

使用 `writeDatagram()` 函数向一个目标用户发送消息时，需要指定目标地址和端口。

在广播消息时，只需将目标地址更换为一个特殊地址，即广播地址 `QHostAddress::Broadcast`，一般是 255.255.255.255。

QUdpSocket 发送的数据报是 `QByteArray` 类型的字节数组，数据报的长度一般不超过 512 字节。数据报的内容可以是文本字符串，也可以自定义格式的二进制数据，文本字符串无需以换行符结束。

QUdpSocket 接收到数据报后发射 `readyRead()` 信号，在关联的槽函数 `onSocketReadyRead()` 里读取缓冲区的数据报，代码如下：

```

void MainWindow::onSocketReadyRead()
{ //读取收到的数据报
    while(udpSocket->hasPendingDatagrams())
    {
        QByteArray datagram;
        datagram.resize(udpSocket->pendingDatagramSize());
        QHostAddress peerAddr;
        quint16 peerPort;
        udpSocket->readDatagram(datagram.data(), datagram.size(), &peerAddr,&peerPort);
        QString str=datagram.data();
        QString peer="[From "+peerAddr.toString()+" ":" "+QString::number(peerPort)+"] ";
        ui->plainTextEdit->appendPlainText(peer+str);
    }
}

```

`hasPendingDatagrams()` 表示是否有待读取的传入数据报。

`pendingDatagramSize()` 返回待读取数据报的字节数。

`readDatagram()` 函数用于读取数据报的内容，其函数原型为：

```

qint64 QUdpSocket::readDatagram(char *data, qint64 maxSize, QHostAddress *address =
Q_NULLPTR, quint16 *port = Q_NULLPTR)

```

输入参数 `data` 和 `maxSize` 是必须的，表示最多读取 `maxSize` 字节的数据到变量 `data` 里。`address` 和 `port` 变量是可选的，用于获取数据报来源的地址和端口。上面的代码中使用了完整的参数形式，



从而可以获得数据报来源的地址 `peerAddr` 和端口 `peerPort`。如果无需获取来源地址和端口，可以采用简略形式，即：

```
udpSocket->readDatagram(datagram.data(), datagram.size());
```

读取的数据报内容是 `QByteArray` 字节数组，因为本程序只是传输字符串，所以简单地将其转换为字符串即可。如果传输的是自定义格式的字符串或二进制数据，需要对接收到的数据进行解析。

### 14.3.3 UDP 组播

#### 1. UDP 组播的特性

图 14-7 的示意图简单表示了组播的原理。UDP 组播是主机之间“一对一组”的通信模式，当多个客户端加入由一个组播地址定义的多播组之后，客户端向组播地址和端口发送的 UDP 数据报，组内成员都可以接收到，其功能类似于 QQ 群。

组播报文的目的地使用 D 类 IP 地址，D 类地址不能出现在 IP 报文的源 IP 地址字段。用同一个 IP 多播地址接收多播数据报的所有主机构成了一个组，称为多播组（或组播组）。所有的信息接收者都加入到一个组内，并且一旦加入之后，流向组地址的数据报立即开始向接收者传输，组中的所有成员都能接收到数据报。组中的成员是动态的，主机可以在任何时间加入和离开组。

所以，采用 UDP 组播必须使用一个组播地址。组播地址是 D 类 IP 地址，有特定的地址段。多播组可以是永久的也可以是临时的。多播组地址中，有一部分由官方分配，称为永久多播组。永久多播组保持不变的是它的 IP 地址，组中的成员构成可以发生变化。永久多播组中成员的数量可以是任意的，甚至可以为零。那些没有保留下来的供永久多播组使用的 IP 组播地址，可以被临时多播组利用。关于组播 IP 地址，有如下的一些约定：

- 224.0.0.0~224.0.0.255 为预留的组播地址（永久组地址），地址 224.0.0.0 保留不做分配，其他地址供路由协议使用；
- 224.0.1.0~224.0.1.255 是公用组播地址，可以用于 Internet；
- 224.0.2.0~238.255.255.255 为用户可用的组播地址（临时组地址），全网范围内有效；
- 239.0.0.0~239.255.255.255 为本地管理组播地址，仅在特定的本地范围内有效。

所以，若是在家庭或办公室局域网内测试 UDP 组播功能，可以使用的组播地址范围是 239.0.0.0~239.255.255.255。

`QUdpSocket` 支持 UDP 组播，`joinMulticastGroup()` 函数使主机加入一个多播组，`leaveMulticastGroup()` 函数使主机离开一个多播组，UDP 组播的特点是使用组播地址，其他的端口绑定、数据报收发等功能的实现与单播 UDP 完全相同。

#### 2. UDP 组播实例程序的功能

设计一个 UDP 组播实例程序 `Samp14_4`，在两台计算机上分别运行，进行组播通信。图 14-10 是运行于主机 192.168.1.104 上的程序，图 14-11 是运行于主机 192.168.1.106 上的程序。两个主机上的程序都加入地址为 239.255.43.21 的多播组，绑定端口 35320 进行通信。

从图 14-10 和图 14-11 可以看到，两个 `Samp14_4` 程序都可以发送和接收组播数据报，且在自己主机上发出的数据报，自己也可以接收到。



图 14-10 主机 A 上运行的 Samp14\_4 程序 (主机地址 192.168.1.104)



图 14-11 主机 B 上运行的 Samp14\_4 程序 (主机地址 192.168.1.106)

### 3. 组播功能的程序实现

程序的主窗口是基于 QMainWindow 的类 MainWindow, 界面由 UI 设计器设计, 其类定义如下 (忽略 UI 设计器生成的槽函数):

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QLabel *LabSocketState;
    QUdpSocket *udpSocket;
    QHostAddress groupAddress; //组播地址
    QString getLocalIP(); //获取本机 IP 地址
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    //自定义槽函数
    void onSocketStateChange(QAbstractSocket::SocketState socketState);
    void onSocketReadyRead(); //读取 socket 传入的数据
private:
    Ui::MainWindow *ui;
};
```

其中定义了一个 QHostAddress 类型变量 groupAddress, 用于记录组播地址。下面是 MainWindow 的构造函数的代码:

```

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    LabSocketState=new QLabel("Socket 状态: ");
    LabSocketState->setMinimumWidth(200);
    ui->statusBar->addWidget(LabSocketState);
    QString localIP=getLocalIP();//本地主机名
    this->setWindowTitle(this->windowTitle()+"----本机 IP: "+localIP);
    udpSocket=new QUdpSocket(this);
    udpSocket->setSocketOption(QAbstractSocket::MulticastTtlOption,1);

    connect(udpSocket,SIGNAL(stateChanged(QAbstractSocket::SocketState)),
        this,SLOT(onSocketStateChange(QAbstractSocket::SocketState)));
    onSocketStateChange(udpSocket->state());
    connect(udpSocket,SIGNAL(readyRead()), this,SLOT(onSocketReadyRead()));
}

```

其中使用了 `QUdpSocket::setSocketOption()` 函数, 对 socket 进行参数设置。

```
udpSocket->setSocketOption(QAbstractSocket::MulticastTtlOption,1);
```

将 socket 的 `QAbstractSocket::MulticastTtlOption` 值设置为 1。MulticastTtlOption 是 UDP 组播的数据报的生存期, 数据报每跨 1 个路由会减 1。缺省值为 1, 表示多播数据报只能在同一路由下的局域网内传播。

要进行 UDP 组播通信, UDP 客户端必须先加入 UDP 多播组, 也可以随时退出多播组。主窗口上的“加入组播”和“退出组播”按钮的代码如下:

```

void MainWindow::on_actStart_triggered()
{
    //加入组播
    QString IP=ui->comboIP->currentText();
    groupAddress=QHostAddress(IP);//多播组地址
    quint16 groupPort=ui->spinPort->value();//端口
    if (udpSocket->bind(QHostAddress::AnyIPv4, groupPort, QUdpSocket::ShareAddress))
    {
        udpSocket->joinMulticastGroup(groupAddress); //加入多播组
        ui->plainTextEdit->appendPlainText("***加入组播成功");
        ui->plainTextEdit->appendPlainText("***组播地址 IP: "+IP);
        ui->plainTextEdit->appendPlainText("***绑定端口: "+QString::number(groupPort));
        ui->actStart->setEnabled(false);
        ui->actStop->setEnabled(true);
        ui->comboIP->setEnabled(false);
    }
    else
        ui->plainTextEdit->appendPlainText("***绑定端口失败");
}

void MainWindow::on_actStop_triggered()
{
    //退出组播
    udpSocket->leaveMulticastGroup(groupAddress); //退出组播
    udpSocket->abort(); //解除绑定
    ui->actStart->setEnabled(true);
    ui->actStop->setEnabled(false);
    ui->comboIP->setEnabled(true);
    ui->plainTextEdit->appendPlainText("***已退出组播,解除端口绑定");
}

```

加入组播之前，必须先绑定端口，绑定端口的语句是：

```
udpSocket->bind(QHostAddress::AnyIPv4, groupPort, QUdpSocket::ShareAddress)
```

这里指定地址为 `QHostAddress::AnyIPv4`，端口为多播组统一的一个端口。

使用 `QUdpSocket::joinMulticastGroup()` 函数加入多播组，即：

```
udpSocket->joinMulticastGroup(groupAddress);
```

多播组地址 `groupAddress` 由界面上的组合框里输入。注意，局域网内的组播地址的范围是 239.0.0.0~239.255.255.255，绝对不能使用本机地址作为组播地址。

退出多播组，使用 `QUdpSocket::leaveMulticastGroup()` 函数，即：

```
udpSocket->leaveMulticastGroup(groupAddress);
```

加入多播组后，发送组播数据报也是使用 `writeDatagram()` 函数，只是目标地址使用的是组播地址，在 `readyRead()` 信号的槽函数里用 `readDatagram()` 读取数据报。下面是发送和读取数据报的代码：

```
void MainWindow::on_btnMulticast_clicked()
{
    //发送组播消息
    quint16 groupPort=ui->spinPort->value();
    QString msg=ui->editMsg->text();
    QByteArray datagram=msg.toUtf8();
    udpSocket->writeDatagram(datagram,groupAddress,groupPort);
    ui->plainTextEdit->appendPlainText("[multicast] "+msg);
    ui->editMsg->clear();
    ui->editMsg->setFocus();
}

void MainWindow::onSocketReadyRead()
{
    //读取数据报
    while(udpSocket->hasPendingDatagrams())
    {
        QByteArray datagram;
        datagram.resize(udpSocket->pendingDatagramSize());
        QHostAddress peerAddr;
        quint16 peerPort;
        udpSocket->readDatagram(datagram.data(), datagram.size(), &peerAddr, &peerPort);
        QString str=datagram.data();
        QString peer="[From "+peerAddr.toString()+": "+QString::number(peerPort)+"] ";
        ui->plainTextEdit->appendPlainText(peer+str);
    }
}
```

## 14.4 基于 HTTP 协议的网络应用程序

### 14.4.1 实现高层网络操作的类

Qt 网络模块提供一些类实现 OSI 7 层网络模型中高层的网络协议，如 HTTP、FTP、SNMP 等，这些类主要是 `QNetworkRequest`、`QNetworkReply` 和 `QNetworkAccessManager`。

`QNetworkRequest` 类通过一个 URL 地址发起网络协议请求，也保存网络请求的信息，目前支



持 HTTP、FTP 和局部文件 URLs 的下载或上传。

QNetworkAccessManager 类用于协调网络操作。在 QNetworkRequest 发起一个网络请求后，QNetworkAccessManager 类负责发送网络请求，创建网络响应。

QNetworkReply 类表示网络请求的响应。由 QNetworkAccessManager 在发送一个网络请求后创建一个网络响应。QNetworkReply 提供的信号 finished()、readyRead() 和 downloadProgress() 可以监测网络响应的执行情况，执行相应操作。

QNetworkReply 是 QIODevice 的子类，所以 QNetworkReply 支持流读写功能，也支持异步或同步工作模式。

## 14.4.2 基于 HTTP 协议的网络文件下载

基于上述三个类，设计一个基于 HTTP 协议的网络文件下载程序，实例程序名称 samp14\_5，图 14-12 是程序运行下载文件时的界面。

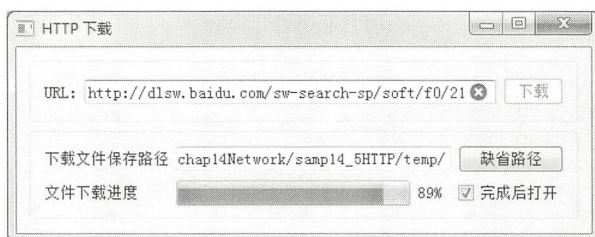


图 14-12 基于 HTTP 的文件下载

在 URL 地址编辑框里输入一个网络文件 URL 地址，设置下载文件保存路径后，单击“下载”按钮就可以开始下载文件到设置的目录下。进度条可以显示文件下载进度，下载完成后还可以用缺省的软件打开下载的文件。URL 里的 HTTP 地址可以是任何类型的文件，如 html、pdf、doc、exe 等。

实例 samp14\_5 主界面是基于 QMainWindow 的窗口类 MainWindow，使用 UI 设计器设计界面，删除了主窗口上的工具栏和状态栏。MainWindow 类的定义如下：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QNetworkAccessManager networkManager; // 网络管理
    QNetworkReply *reply; // 网络响应
    QFile *downloadedFile; // 下载保存的临时文件
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    // 自定义槽函数
    void on_finished();
    void on_readyRead();
    void on_downloadProgress(qint64 bytesRead, qint64 totalBytes);
private:
```

```

    Ui::MainWindow *ui;
};

```

要下载文件，先在窗口上的 URL 编辑框里输入下载地址（可以使用 Ctrl+V 组合键粘贴 URL 地址），再设置下载文件保存的目录。单击“缺省路径”按钮会在程序的当前目录下创建一个临时文件夹，代码如下：

```

void MainWindow::on_btnDefaultPath_clicked()
{
    //缺省路径 按钮
    QString curPath=QDir::currentPath();
    QDir dir(curPath);
    QString sub="temp";
    dir.mkdir(sub);
    ui->editPath->setText(curPath+"/"+sub+"/");
}

```

输入这些设置后，单击“下载”按钮开始下载过程，“下载”按钮的响应代码如下：

```

void MainWindow::on_btnDownload_clicked()
{
    //开始下载
    QString urlSpec = ui->editURL->text().trimmed();
    if (urlSpec.isEmpty())
    {
        QMessageBox::information(this, "错误", "请指定需要下载的 URL");
        return;
    }

    QUrl newUrl = QUrl::fromUserInput(urlSpec);
    if (!newUrl.isValid())
    {
        QMessageBox::information(this, "错误",
            QString("无效 URL: %1 \n 错误信息: %2").arg(urlSpec, newUrl.errorString()));
        return;
    }

    QString tempDir =ui->editPath->text().trimmed();
    if (tempDir.isEmpty())
    {
        QMessageBox::information(this, "错误", "请指定保存下载文件的目录");
        return;
    }

    QString fullFileName =tempDir+newUrl.fileName();
    if (QFile::exists(fullFileName))
        QFile::remove(fullFileName);

    downloadedFile =new QFile(fullFileName);
    if (!downloadedFile->open(QIODevice::WriteOnly))
    {
        QMessageBox::information(this, "错误", "临时文件打开错误");
        return;
    }
    ui->btnDownload->setEnabled(false);
    reply = networkManager.get(QNetworkRequest(newUrl));
    connect(reply, SIGNAL(finished()), this, SLOT(on_finished()));
    connect(reply, SIGNAL(readyRead()), this, SLOT(on_readyRead()));
    connect(reply, SIGNAL(downloadProgress(qint64,qint64)),
        this, SLOT(on_downloadProgress(qint64,qint64)));
}

```

代码在读取 URL 地址后，将其转换为一个 QUrl 类变量 newUrl，并检查其有效性，再检查临

时文件目录，创建临时文件 `downloadedFile`。

这些准备好之后，用 `QNetworkAccessManager` 发布网络请求，请求下载 URL 地址表示的文件，并创建网络响应，关键代码如下：

```
reply = networkManager.get(QNetworkRequest(newUrl));
```

`reply` 为网络响应，将其 3 个信号与相关的自定义槽函数相关联，实现相应的操作。这 3 个槽函数的代码如下：

```
void MainWindow::on_readyRead()
{ // 读取下载的数据
    downloadedFile->write(reply->readAll());
}
void MainWindow::on_downloadProgress(qint64 bytesRead, qint64 totalBytes)
{ // 下载进程
    ui->progressBar->setMaximum(totalBytes);
    ui->progressBar->setValue(bytesRead);
}
void MainWindow::on_finished()
{ // 网络响应结束
    QFileInfo fileInfo;
    fileInfo.setFile(downloadedFile->fileName());
    downloadedFile->close();
    delete downloadedFile;
    downloadedFile = Q_NULLPTR;
    reply->deleteLater();
    reply = Q_NULLPTR;
    if (ui->checkOpen->isChecked())
        QDesktopServices::openUrl(QUrl::fromLocalFile(
            fileInfo.absoluteFilePath()));
    ui->btnDownload->setEnabled(true);
}
```

在缓冲区有新下载的数据等待读取时，会发射 `readyRead()` 信号，槽函数 `on_readyRead()` 读取下载缓冲区的数据到临时文件。

`downloadProgress()` 是表示网络操作进度的信号，传递 `bytesRead` 和 `totalBytes` 两个参数，表示已读取字节数和总的字节数；`on_downloadProgress()` 槽函数将这两个参数用于进度条的显示，可以显示下载进度。

`finished()` 信号在下载结束后发射，槽函数 `on_finished()` 的功能是关闭临时文件，删除文件变量和网络响应变量的。然后用 `QDesktopServices::openUrl()` 函数调用缺省的应用软件打开下载的文件，例如，如果下载的是一个 PDF 文件，会自动用相关联的 PDF 阅读器软件打开此文件。

多媒体功能指的主要是计算机的音频和视频的输入、输出、显示和播放等功能，Qt 的多媒体模块为音频和视频播放、录音、摄像头拍照和录像等提供支持，甚至还提供数字收音机的支持。本章将介绍 Qt 多媒体模块的功能和使用。

## 15.1 Qt 多媒体模块功能概述

Qt 多媒体模块提供了很多类，可以实现如下的一些功能：

- 访问原始音频设备进行输入或输出；
- 低延迟播放音效文件，如 WAV 文件；
- 使用播放列表播放压缩的音频和视频文件，如 mp3、wmv 等；
- 录制声音并且压制文件；
- 使用摄像头进行预览、拍照和视频录制；
- 音频文件解码到内存进行处理；
- 录制音频或视频时，访问其视频帧或音频缓冲区；
- 数字广播调谐和收听。

要在 C++ 项目中使用 Qt 多媒体模块，需要在项目配置文件中添加如下一行语句：

```
Qt += multimedia
```

如果在项目中使用视频播放功能，还需要加入下面的一行，以使用 QVideoWidget 或 QGraphicsVideoItem 进行视频播放。

```
Qt += multimediawidgets
```

Qt 多媒体模块包括多个类，表 15-1 是一些典型的多媒体应用所需要用到的主要的类。

表 15-1 各类多媒体功能用到的类

应用功能	用到的类
播放压缩音频（MP3、AAC 等）	QMediaPlayer, QMediaPlaylist
播放音效文件（WAV 文件）	QSoundEffect, QSound
播放低延迟的音频	QAudioOutput
访问原始音频输入数据	QAudioInput
录制编码的音频数据	QAudioRecorder
发现音频设备	QAudioDeviceInfo
视频播放	QMediaPlayer, QVideoWidget, QGraphicsVideoItem



续表

应用功能	用到的类
视频处理	QMediaPlayer, QVideoFrame, QAbstractVideoSurface
摄像头取景框	QCamera, QVideoWidget, QGraphicsVideoItem
取景框预览处理	QCamera, QAbstractVideoSurface, QVideoFrame
摄像头拍照	QCamera, QCameraImageCapture
摄像头录像	QCamera, QMediaRecorder
收听数字广播	QRadioTuner, QRadioData

利用 Qt 多媒体模块提供的各种类, 可以实现一般的音频、视频的输入和输出。这在一些实际应用中是需要的, 如语音识别需要录制音频并对音频数据进行处理, 车牌自动识别需要先拍照然后进行图像处理。

本章将介绍音频的播放和录制、视频播放、摄像头拍照与录像等功能的实现。数字广播需要具有数字调频功能的硬件设备, 一般用户没有这样的硬件设备, 本书就不予介绍了。

## 15.2 音频播放

### 15.2.1 使用 QMediaPlayer 播放音乐文件

#### 1. 音频播放器实例程序

QMediaPlayer 可以播放经过压缩的音频或视频文件, 如 mp3、mp4、wmv 等文件, QMediaPlayer 可以播放单个文件, 也可以和 QMediaPlaylist 类结合, 对一个播放列表进行播放。所以使用 QMediaPlayer 和 QMediaPlaylist 可以轻松地设计一个自己的音乐或视频播放器。

QMediaPlayer 的主要公共函数和槽函数见表 15-2 (省略了函数中的 const 关键字和缺省参数)。

表 15-2 QMediaPlayer 的主要函数

函数原型	功能描述
qint64 duration()	当前文件播放时间总长, 单位 ms
void setPosition(qint64 position)	设置当前文件播放位置, 单位 ms
void setMuted(bool muted)	设置是否静音
bool isMuted()	返回是否静音的状态, true 表示静音
void setPlaylist(QMediaPlaylist *playlist)	设置播放列表
QMediaPlaylist* playlist()	返回设置的播放列表
State state()	返回播放器当前的状态
void setVolume(int volume)	设置播放音量, 0 至 100 之间
void setPlaybackRate(qreal rate)	设置播放速度, 缺省为 1, 表示正常速度
void setMedia(QMediaContent &media)	设置播放媒体文件
QMediaContent currentMedia()	返回当前播放的媒体文件
void play()	开始播放
void pause()	暂停播放
void stop()	停止播放

使用 QMediaPlayer 播放媒体文件时, 有几个有用的信号可以反映播放状态或文件信息。

- stateChanged(QMediaPlayer::State state)信号在调用 play()、pause()和 stop()函数时发射, 反映播放器当前的状态。枚举类型 QMediaPlayer::State 有 3 种取值, 表示播放器的状态:

QMediaPlayer::StoppedState, 停止状态;

QMediaPlayer::PlayingState, 正在播放;

QMediaPlayer::PausedState, 暂停播放状态。

- durationChanged(qint64 duration)信号在文件的时间长度变化时发射, 一般在切换播放文件时发射。
- positionChanged(qint64 position)当前文件播放位置变化时发射, 可以反映文件播放进度。

QMediaPlayer 可以通过 setMedia()函数设置播放单个文件, 也可以通过 setPlaylist()函数设置一个 QMediaPlaylist 类实例表示的播放列表, 对列表文件进行播放, 并且自动播放下一个文件, 或循环播放等。QMediaPlayer 播放的文件可以是本地文件, 也可以是网络上的文件。

QMediaPlaylist 记录播放媒体文件信息, 可以添加、移除文件, 也可以设置循环播放形式, 在列表文件中自动切换文件。在当前播放文件切换时会发射 currentIndexChanged() 信号和 currentMediaChange()信号。

使用 QMediaPlayer 和 QMediaPlaylist 的这些功能, 可以实现一个完整功能的音乐播放器。图 15-1 是使用 QMediaPlayer 和 QMediaPlaylist 实现的一个音乐播放器实例程序 samp15\_1, 它实现了一个音乐播放器的基本功能。

## 2. 界面设计与主窗口类的定义

实例 samp15\_1 是一个界面基于 QMainWindow 的应用程序, 主窗口上删除了菜单栏、工具栏和状态栏, 界面采用 UI 设计器设计, 中间是一个 QListWidget 组件显示播放列表的文件名。其他的界面组件和布局设计不再赘述。



图 15-1 使用 QMediaPlayer 和 QmediaPlaylist 实现的音乐播放器

下面是主窗口类的定义 (省略了 UI 设计器自动生成的界面组件的槽函数):

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QMediaPlayer    *player;//播放器
    QMediaPlaylist  *playlist;//播放列表
    QString         durationTime;//总长度
    QString         positionTime;//当前播放到的位置
public:
    explicit MainWindow(QWidget *parent = 0);
private slots:
    //自定义槽函数
    void onStateChanged(QMediaPlayer::State state);
    void onPlaylistChanged(int position);
    void onDurationChanged(qint64 duration);
    void onPositionChanged(qint64 position);
private:
    Ui::MainWindow *ui;
};
```

主要是定义了 4 个私有变量, 4 个自定义槽函数。

- `onStateChanged()`在播放器播放状态变化时发射，以更新界面上的“播放”“暂停”“停止”按钮的使能状态。
- `onPlaylistChanged()`在播放列表的当前文件变化时发射，用以更新界面上显示当前媒体文件名。
- `onDurationChanged()`在文件时长变化时发射，用于更新界面上文件时间长度的显示。
- `onPositionChanged()`在当前文件播放位置变化时发射，用于更新界面上的播放进度显示。

下面是 `MainWindow` 构造函数的代码，主要功能是创建 `player` 和 `playlist`，然后进行信号与自定义槽函数的关联。

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setUpUi(this);
    player = new QMediaPlayer(this); //播放器
    playlist = new QMediaPlaylist(this); //播放列表
    playlist->setPlaybackMode(QMediaPlaylist::Loop); //循环模式
    player->setPlaylist(playlist);

    connect(player, SIGNAL(stateChanged(QMediaPlayer::State)),
            this, SLOT(onStateChanged(QMediaPlayer::State)));
    connect(player, SIGNAL(positionChanged(qint64)),
            this, SLOT(onPositionChanged(qint64)));
    connect(player, SIGNAL(durationChanged(qint64)),
            this, SLOT(onDurationChanged(qint64)));
    connect(playlist, SIGNAL(currentIndexChanged(int)),
            this, SLOT(onPlaylistChanged(int)));
}
```

下面是 4 个自定义槽函数的代码：

```
void MainWindow::onStateChanged(QMediaPlayer::State state)
{ //播放器状态变化，更新按钮状态
    ui->btnPlay->setEnabled(!(state==QMediaPlayer::PlayingState));
    ui->btnPause->setEnabled(state==QMediaPlayer::PlayingState);
    ui->btnStop->setEnabled(state==QMediaPlayer::PlayingState);
}

void MainWindow::onPlaylistChanged(int position)
{ //播放列表变化，更新当前播放文件名显示
    ui->listWidget->setCurrentRow(position);
    QListWidget::Item *item=ui->listWidget->currentItem();
    if (item)
        ui->LabCurMedia->setText(item->text());
}

void MainWindow::onDurationChanged(qint64 duration)
{ //文件时长变化，更新进度显示
    ui->sliderPosition->setMaximum(duration);
    int secs=duration/1000; //秒
    int mins=secs/60; //分钟
    secs=secs % 60; //余数秒
    durationTime=QString::asprintf("%d:%d",mins,secs);
    ui->LabRatio->setText(positionTime+"/"+durationTime);
}

void MainWindow::onPositionChanged(qint64 position)
{ //当前文件播放位置变化，更新进度显示
```

```

if (ui->sliderPosition->isSliderDown()) //正处于手动调整状态, 不处理
    return;
ui->sliderPosition->setSliderPosition(position);
int secs=position/1000;//秒
int mins=secs/60; //分钟
secs=secs % 60;//余数秒
positionTime=QString::asprintf("%d:%d",mins,secs);
ui->LabRatio->setText(positionTime+"/"+durationTime);
}

```

### 3. 播放列表控制

窗口中间以一个 QListWidget 组件显示播放的文件列表, 界面上显示的文件列表与 playlist 存储的文件列表保持同步。

窗口上方的“添加”“移除”“清空”3个按钮的代码如下:

```

void MainWindow::on_btnAdd_clicked()
{ //添加文件
    QString curPath=QDir::homePath();//获取用户目录
    QString dlgTitle="选择音频文件";
    QString filter="音频文件 (*.mp3 *.wav *.wma);;mp3 文件 (*.mp3);;wav 文件 (*.wav);;wma 文件 (*.wma);;所有文件 (*.*)";
    QStringList fileList=QFileDialog::getOpenFileNames(this, dlgTitle, curPath, filter);
    if (fileList.count() < 1)
        return;
    for (int i=0; i<fileList.count(); i++)
    {
        QString aFile=fileList.at(i);
        playlist->addMedia(QUrl::fromLocalFile(aFile)); //添加文件
        QFileInfo fileInfo(aFile);
        ui->listWidget->addItem(fileInfo.fileName()); //添加到界面文件列表
    }
    if (player->state() != QMediaPlayer::PlayingState)
        playlist->setCurrentIndex(0);
    player->play();
}

void MainWindow::on_btnRemove_clicked()
{ //移除一个文件
    int pos=ui->listWidget->currentRow();
    QListWidgetItem *item=ui->listWidget->takeItem(pos);
    delete item; //从 listWidget 里删除
    playlist->removeMedia(pos); //从播放列表里删除
}

void MainWindow::on_btnClear_clicked()
{ //清空列表
    playlist->clear();
    ui->listWidget->clear();
    player->stop();
}

```

用到的 QMediaPlaylist 的主要函数有:

- addMedia() 函数添加一个文件;
- removeMedia() 移除一个文件;
- setCurrentIndex() 设置当前播放文件序号;



- `clear()`清空播放列表。

在播放列表中前移和后移使用 `previous()`和 `next()`函数, 移动时播放列表会发射 `currentIndexChanged()`信号, 从而自动更新界面上 `listWidget` 里的当前条目。

在界面上的 `listWidget` 里双击一个条目时, 切换到播放这个文件, 其实现代码为:

```
void MainWindow::on_listWidget_doubleClicked(const QModelIndex &index)
{ //双击时切换播放文件
    int rowNo=index.row();
    playlist->setCurrentIndex(rowNo);
    player->play();
}
```

#### 4. 播放控制

播放、暂停或停止播放器, 只需调用 `QMediaPlayer` 相应函数即可, 界面上 3 个按钮的代码如下:

```
void MainWindow::on_btnPlay_clicked()
{ //播放
    if (playlist->currentIndex()<0)
        playlist->setCurrentIndex(0);
    player->play();
}
void MainWindow::on_btnPause_clicked()
{ //暂停播放
    player->pause();
}
void MainWindow::on_btnStop_clicked()
{ //停止播放
    player->stop();
}
```

播放状态变化时会发射 `stateChanged()`信号, 在关联的自定义槽函数 `onStateChanged()`里更新 3 个按钮的使能状态。

音量控制由一个“静音”按钮和音量滑动条控制, 相关代码如下:

```
void MainWindow::on_btnSound_clicked()
{ //静音控制
    bool mute=player->isMuted();
    player->setMuted(!mute);
    if (mute)
        ui->btnSound->setIcon(QIcon(":/images/images/volumn.bmp"));
    else
        ui->btnSound->setIcon(QIcon(":/images/images/mute.bmp"));
}
void MainWindow::on_sliderVolumn_valueChanged(int value)
{ //调整音量
    player->setVolume(value);
}
```

文件播放进度条在 `onDurationChanged()`和 `onPositionChanged()`两个自定义槽函数里会更新, 显示当前文件播放进度。当拖动滑动条的滑块可以设置文件播放位置, 代码如下:

```
void MainWindow::on_sliderPosition_valueChanged(int value)
{ //文件进度调控
    player->setPosition(value);
}
```

## 15.2.2 使用 QSoundEffect 和 QSound 播放音效文件

QSoundEffect 用于播放低延迟的音效文件，如无压缩的 WAV 文件，用于实现一些音效效果，如按键音、提示音等。使用 QSoundEffect 播放音效文件的示例代码如下：

```
QSoundEffect effect;
effect.setSource(QUrl::fromLocalFile("engine.wav"));
effect.setLoopCount(3);
effect.setVolume(1);
effect.play();
```

QSoundEffect 不仅可以播放本地文件，还可以播放网络文件。

还有一个类 QSound 只能播放本地 WAV 文件，而且是异步方式播放。可以直接使用 QSound 的静态函数播放 WAV 文件，如：

```
QSound::play("mysounds/bells.wav");
```

## 15.3 音频输入

音频输入可以使用 QAudioRecorder 或 QAudioInput 两个类实现。QAudioRecorder 是高层次的实现，输入的音频数据直接保存为文件，也可以通过 QAudioProbe 访问原始的音频数据。QAudioInput 是低层次的实现，直接控制音频输入设备的参数，并将音频录制数据写入一个流设备。

### 15.3.1 使用 QAudioRecorder 录制音频

#### 1. 使用 QAudioRecorder 录制音频

QAudioRecorder 是用于录制音频的类，它从 QMediaRecorder 类继承而来，只需要较少的代码，就可以实现音频录制并存储到文件。图 15-2 是使用 QAudioRecorder 录制音频文件的实例程序 samp15\_3 运行时界面。



图 15-2 实例 samp15\_3 运行时界面

QAudioRecorder 需要使用一个 QAudioEncoderSettings 类型的变量进行输入音频设置, 主要是编码格式、采样率、通道数、音频质量等高级设置, 图 15-2 窗口左侧是音频输入设置。

设置一个输出保存文件后就可以使用 QAudioRecorder 录制文件, 录制的数据会自动保存到文件里。音频输入设备会根据音频设置自动确定底层的采样参数, 使用 QAudioProbe 类可以获取音频输入缓冲区的参数和原始数据。图 15-2 窗口右侧显示了音频输入缓冲区的数据参数, 包括缓冲区字节数、帧数、采样数、采样字长、采样率等, 通过这些参数就可以从缓冲区读取原始的音频数据。

## 2. QAudioRecorder 录音功能的实现

samp15\_3 的主窗口从 QMainWindow 继承, 界面采用 UI 设计器设计, 界面组件和布局的设计不再详述。主窗口类 MainWindow 的定义如下 (省略了 UI 生成的界面组件的槽函数定义):

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QAudioRecorder *recorder; //音频录音
    QAudioProbe *probe; //探测器
public:
    explicit MainWindow(QWidget *parent = 0);
private slots:
    //自定义槽函数
    void onStateChanged(QMediaRecorder::State state);
    void onDurationChanged(qint64 duration);
    void processBuffer(const QAudioBuffer& buffer);
private:
    Ui::MainWindow *ui;
};
```

QAudioRecorder 类型变量 recorder 用于录音, QAudioProbe 类型变量 probe 用于探测缓冲区数据。

还定义了 3 个槽函数, 用于与 recorder 和 probe 的信号进行关联。在窗口的构造函数里创建变量和进行信号与槽的关联。下面是窗口构造函数的代码:

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    recorder = new QAudioRecorder(this);
    connect(recorder, SIGNAL(stateChanged(QMediaRecorder::State)),
            this, SLOT(onStateChanged(QMediaRecorder::State)));
    connect(recorder, SIGNAL(durationChanged(qint64)), this,
            SLOT(onDurationChanged(qint64)));

    probe = new QAudioProbe; //探测器
    connect(probe, SIGNAL(audioBufferProbed(QAudioBuffer)),
            this, SLOT(processBuffer(QAudioBuffer)));
    probe->setSource(recorder);

    if (recorder->defaultAudioInput().isEmpty())
        return; //无音频输入设备
    foreach (const QString &device, recorder->audioInputs())
        ui->comboDevices->addItem(device); //音频输入设备列表
```

```

foreach (const QString &codecName, recorder->supportedAudioCodecs())
    ui->comboCodec->addItem(codecName); //支持的音频编码
foreach (int sampleRate, recorder->supportedAudioSampleRates())
    ui->comboSampleRate->addItem(QString::number(sampleRate)); //采样率
//channels
ui->comboChannels->addItem("1");
ui->comboChannels->addItem("2");
ui->comboChannels->addItem("4");
//quality
ui->sliderQuality->setRange(0, int(QMultimedia::VeryHighQuality));
ui->sliderQuality->setValue(int(QMultimedia::NormalQuality));
//bitrates:
ui->comboBitrate->addItem("32000");
ui->comboBitrate->addItem("64000");
ui->comboBitrate->addItem("96000");
ui->comboBitrate->addItem("128000");
}

```

槽函数 `onStateChanged()` 与 `recorder` 的 `stateChanged()` 信号关联, 在 `recorder` 的状态变化时控制界面上“录音”“暂停”“停止”等按钮的使能状态。

槽函数 `onDurationChanged()` 与 `recorder` 的 `durationChanged()` 信号关联, 该信号在录制持续时间变化时发射, 以更新界面上的时间显示。下面是这两个槽函数的代码:

```

void MainWindow::onStateChanged(QMediaRecorder::State state)
{ //录音状态变化
    ui->actRecord->setEnabled(state!=QMediaRecorder::RecordingState);
    ui->actPause->setEnabled(state==QMediaRecorder::RecordingState);
    ui->actStop->setEnabled(state==QMediaRecorder::RecordingState);
    ui->btnGetFile->setEnabled(state==QMediaRecorder::StoppedState);
    ui->editOutputFile->setEnabled(state==QMediaRecorder::StoppedState);
}

void MainWindow::onDurationChanged(qint64 duration)
{ //录音持续时间变化
    ui->LabPassTime->setText(QString("已录制 %1 秒").arg(duration / 1000));
}

```

探测器 `probe` 通过 `setSource()` 指定探测对象, 这里 `probe` 的探测对象是 `recorder`。

```
probe->setSource(recorder);
```

`QAudioProbe` 也可以在音频播放时进行缓冲区探测, 也就是可以指定一个 `QMediaPlayer` 对象作为探测对象。

槽函数 `processBuffer()` 与 `probe` 的信号 `audioBufferProbed()` 关联, 这个信号会传递一个 `QAudioBuffer` 类型的变量, 这个变量里存储了缓冲区的信息和音频原始数据。

构造函数的后部分对界面上录音设置的组件的内容进行初始化, 用到 `QAudioRecorder` 类的如下一些函数:

- `defaultAudioInput()` 函数获取缺省的音频输入设备名称;
- `audioInputs()` 函数获取音频输入设备列表;
- `supportedAudioCodecs()` 函数获取支持的音频编码列表;
- `supportedAudioSampleRates()` 获取支持的音频采样率列表。



选择一个录音输出文件后，就可以通过界面上“录音”“暂停”“停止”三个按钮进行录音控制，三个按钮代码如下：

```
void MainWindow::on_actRecord_triggered()
{ //开始录音
    if (recorder->state() == QMediaRecorder::StoppedState) //已停止，重新设置
    {
        QString selectedFile=ui->editOutputFile->text().trimmed();
        if (selectedFile.isEmpty())
        {
            QMessageBox::critical(this, "错误", "请先设置录音输出文件");
            return;
        }
        if (QFile::exists(selectedFile))
            if (!QFile::remove(selectedFile))
            {
                QMessageBox::critical(this, "错误", "所设置录音输出文件无法删除");
                return;
            }
        recorder->setOutputLocation(QUrl::fromLocalFile(selectedFile));
        recorder->setAudioInput(ui->comboDevices->currentText()); //输入设备
        QAudioEncoderSettings settings; //音频编码设置
        settings.setCodec(ui->comboCodec->currentText()); //编码
        settings.setSampleRate(ui->comboSampleRate->currentText().toInt());
        settings.setBitRate(ui->comboBitrate->currentText().toInt()); //比特率
        settings.setChannelCount(ui->comboChannels->currentText().toInt());
        settings.setQuality(QMultimedia::EncodingQuality(
            ui->sliderQuality->value()));
        if (ui->radioQuality->isChecked()) //编码模式为固定品质
            settings.setEncodingMode(QMultimedia::ConstantQualityEncoding);
        else
            settings.setEncodingMode(QMultimedia::ConstantBitRateEncoding);
        recorder->setAudioSettings(settings); //音频设置
    }
    recorder->record();
}

void MainWindow::on_actPause_triggered()
{ //暂停
    recorder->pause();
}

void MainWindow::on_actStop_triggered()
{ //停止
    recorder->stop();
}
```

开始、暂停和停止录音只需调用 QAudioRecorder 的 record()、pause()和 stop()函数，这会导致 QAudioRecorder 的 state()发生变化，并发射 stateChanged()信号，在关联的槽函数 onStateChanged()里更新界面按钮的使能状态。

在“录音”按钮的代码里，如果是从停止状态单击“录音”，将会根据界面的输入用 setOutputLocation()设置保存文件；并用 setAudioInput()设置音频输入设备；然后用一个 QaudioEncoderSettings 类型变量 settings 获取录音设置，设置的内容包括：

- setCodec()设置音频编码，如“audio/pcm”是未经编码的音频格式；

- `setSampleRate()`设置采样率, 如 8000Hz 是最低的采样率, 44100Hz 是一般 CD、MP3 的采样率, 96000Hz 是高清晰音轨使用的采样率;
- `setChannelCount()`设置通道数, 常见的有单声道、立体声(双声道)和四声环绕(四声道);
- `setBitRate()`设置比特率, 若用 `QAudioEncoderSettings::setEncodingMode()`设置, 编码模式为 `QMultimedia::ConstantBitRateEncoding`, 则音频输入设备采用固定的比特率采样, 比特率越高, 音质越好, 一般较高音质如 128 kbps;
- `setQuality()`设置录音质量, 传递的参数是表示录音质量的枚举类型 `QMultimedia::EncodingQuality`, 有“VeryLowQuality”到“VeryHighQuality”五个等级。若编码模式设置为固定质量, 则音频输入设备会根据质量要求自动设置底层的采样率、字长等参数。

在配置好 `settings` 的内容后, 为录音器设置音频参数, 即:

```
recorder->setAudioSettings(settings);
```

### 3. QAudioProbe 获取音频输入缓冲区数据参数

单击“录音”按钮后, 就可以根据设置进行录音, 录音的数据会自动保存到指定的存储文件里。

由于使用了一个 `QAudioProbe` 类型变量 `probe` 进行录音数据的探测, 在录音过程中, `probe` 会在录音的缓冲区更新数据后发射 `audioBufferProbed()`信号。与 `audioBufferProbed()`信号关联的自定义槽函数 `processBuffer()`对缓冲区的数据信息进行查询和显示, 代码如下:

```
void MainWindow::processBuffer(const QAudioBuffer &buffer)
{ //处理探测到的缓冲区
    ui->spin_byteCount->setValue(buffer.byteCount()); //缓冲区字节数
    ui->spin_duration->setValue(buffer.duration()/1000); //缓冲区时长
    ui->spin_frameCount->setValue(buffer.frameCount()); //缓冲区帧数
    ui->spin_sampleCount->setValue(buffer.sampleCount()); //缓冲区采样数
    QAudioFormat audioFormat=buffer.format(); //缓冲区格式
    ui->spin_channelCount->setValue(audioFormat.channelCount()); //通道数
    ui->spin_sampleSize->setValue(audioFormat.sampleSize()); //采样大小
    ui->spin_sampleRate->setValue(audioFormat.sampleRate()); //采样率
    ui->spin_bytesPerFrame->setValue(audioFormat.bytesPerFrame());
    if (audioFormat.byteOrder()==QAudioFormat::LittleEndian)
        ui->edit_byteOrder->setText("LittleEndian"); //字节序
    else
        ui->edit_byteOrder->setText("BigEndian");
    ui->edit_codec->setText(audioFormat.codec()); //编码格式

    if (audioFormat.sampleType()==QAudioFormat::SignedInt) //采样点类型
        ui->edit_sampleType->setText("SignedInt");
    else if (audioFormat.sampleType()==QAudioFormat::UnsignedInt)
        ui->edit_sampleType->setText("UnsignedInt");
    else if (audioFormat.sampleType()==QAudioFormat::Float)
        ui->edit_sampleType->setText("Float");
    else
        ui->edit_sampleType->setText("Unknown");
}
```

信号 `audioBufferProbed()`传递一个 `QAudioBuffer` 类型的参数 `buffer`, 该参数存储了缓冲区的音频采样参数和音频原始数据。通过 `QAudioBuffer::format()`函数可以获得音频格式参数。

```
QAudioFormat audioFormat=buffer.format();
```

`buffer.format()`返回一个 `QAudioFormat` 类型的变量,存储了音频的格式参数信息。如果要使用音频的原始数据,需要对这些参数有所了解,结合图 15-2 显示的内容对 `QAudioFormat` 的一些函数表示的参数作解释。

- `channelCount()`返回音频数据的实际通道数,与前面的音频设置的通道数一致。
- `sampleSize()`返回采样点位数,是指一个采样数据点的量化位数,一般有 8 位、16 位和 32 位。位数越多,声音的分辨率越高,保真度也越高,一般 16 位即可达到 CD 的音频质量。
- `sampleRate()`返回实际的采样频率,一般等于或大于音频输入设置的采样率,也会根据设置的质量要求自动设置实际的采样率。
- `sampleType()`返回采样点格式,是指一个采样点得用什么类型的数据来表示,有无符号整型 (`UnsignedInt`)、有符号整型 (`SignedInt`) 和浮点数 (`Float`)。
- `byteOrder()`返回字节序,分为大端字节序和小端字节序。
- `codec()`返回实际的编码方式。
- `bytesPerFrame()`返回每帧字节数,不同音频编码格式的帧的定义不一样,PCM 的一帧就是各个通道的一次采样数据。

`QAudioBuffer` 还有其他一些函数来说明缓冲区数据的信息,结合图 15-2 对这些参数进行解释如下。

- `frameCount()`返回帧数,对于 PCM 格式编码的音频数据,一帧就是一次采样点,这个函数返回了缓冲区中数据点的帧数,如 320 帧。
- `sampleCount()`返回采样数,采样数=帧数\*通道数,因为是 2 个通道,所以采样数为 640。
- `byteCount()`返回缓冲区字节数,字节数=采样数\*采样字节数,因为采样点位数 8 位,即 1 个字节,所以缓冲区字节数为 640。
- `duration()`返回缓冲区时长,时长由帧数和采样频率决定,图中的缓冲区时长为 40 ms。因为帧数为 320,采样频率为 8000 Hz,所以时长为:

$$\frac{320}{8000} \times 1000 \text{ ms} = 40 \text{ ms}$$

`QAudioBuffer::data()`函数返回缓冲区存储的音频的原始数据,获取这些原始数据,就可以对数据进行分析或处理,如进行语音识别必须先获得这些音频原始数据。

使用 `QAudioRecorder` 进行音频输入时,由于设置的采集参数不同,如要求音频质量不同时,底层的音频采样参数会自动调整。与图 15-2 中的设置相似,只是将音频编码质量设置为最高质量,录音时缓冲区的信息就发生较大的变化,如图 15-3 所示。图中可以看到缓冲区的采样字长变为了 16 位,采样率变为了 96000 Hz,每帧字节数变为了 4 字节。

所以在采用 `QAudioBuffer::data()`读取原始数据时,需要根据 `QAudioBuffer::format()`返回的格式参数以及缓冲区帧数、采样数等参数才能正确读取原始数据,这实现起来比较复杂,本例就不演示原始数据的读取了。





图 15-3 固定品质为最高品质时录音的缓冲区参数

### 15.3.2 使用 QAudioInput 获取音频输入

#### 1. QAudioInput 获取音频输入功能概述

QAudioInput 类提供了接收音频设备输入数据的接口，创建 QAudioInput 对象实例时，需要用两个参数，一个是 QAudioDeviceInfo 类表示的音频设备，一个是 QAudioFormat 表示的音频输入格式。QAudioInput::start() 函数开始音频数据输入时，需要指定一个流设备接收输入的音频数据，如可以指定一个 QFile 表示的文件。

QAudioInput 与 QAudioRecorder 的不同之处如下。

- QAudioInput 创建时指定的 QAudioFormat 将直接作用于音频输入设备，也就是音频输入的数据将直接按照设置的参数进行采样，而 QAudioRecorder 不能直接控制采样字长、采样点类型等底层参数。
- QAudioInput::start(QIODevice \*device) 指定一个 QIODevice 设备作为数据输出对象，可以是文件，也可以是其他从 QIODevice 继承的类。如从 QIODevice 继承一个类，对输入的缓冲区数据进行处理，而不是保存到文件。而 QAudioRecorder 只能指定文件作为保存对象。所以，QAudioInput 可以实现更加底层的音频输入控制。

图 15-4 是使用 QAudioInput 实现的一个音频数据输入并实时显示原始信号波形的实例程序 samp15\_4 的运行界面。

图 15-4 左侧显示的是用 QAudioDeviceInfo 类获取的音频设备，以及设备支持的各种参数，单击“测试音频设置”可以判断音频设备是否支持所设置的采集配置。为了方便读取原始数据，在开始采集时采用固定的设置，即 8000Hz、1 通道、8 位、无符号整数。

窗口右侧是一个 QChart 组件，采用 QLineSeries 作为显示序列。开始采集后，从缓冲区读取的数据将实时显示在图表上。



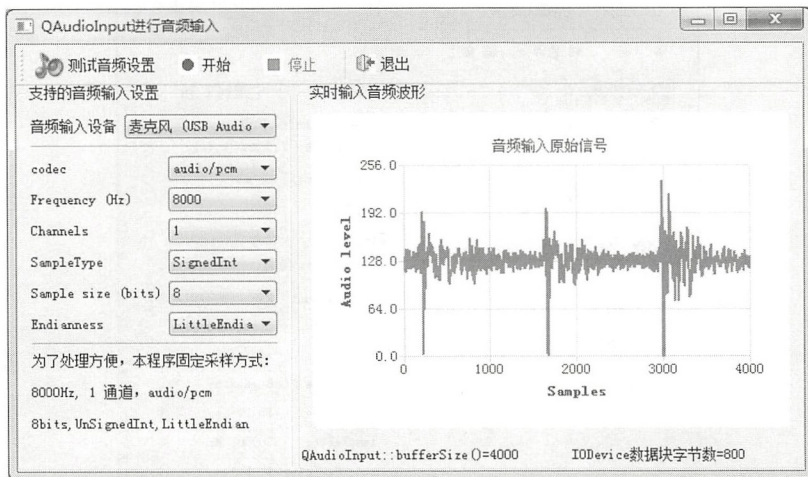


图 15-4 使用 QAudioInput 获取音频输入并显示原始数据波形

## 2. 主窗口定义与初始化

samp15\_4 的主窗口是基于 QMainWindow 的类 MainWindow, 窗口界面设计由 UI 设计器实现。

主窗口类 MainWindow 的定义如下:

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    const qint64 displayPointsCount=4000;
    QLineSeries *lineSeries; //曲线序列
    QList<QAudioDeviceInfo> deviceList; //音频输入设备列表
    QAudioDeviceInfo curDevice; //当前输入设备
    QmyDisplayDevice *displayDevice; //用于显示的 IODevice
    QAudioInput *audioInput; //音频输入设备
    QString SampleTypeString(QAudioFormat::SampleType sampleType);
    QString ByteOrderString(QAudioFormat::Endian endian);
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    //自定义槽函数
    void on_IODevice_UpdateBlockSize(qint64 blockSize);
private:
    Ui::MainWindow *ui;
};
```

这里定义了较多的私有变量, 其中 QmyDisplayDevice 是一个自定义的从 QIODevice 继承的类, 用于读取音频输入缓冲区的数据, 并在图表上显示。其具体实现在后面介绍。

MainWindow 的构造函数代码如下:

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setUpUi(this);
```

```

    setCentralWidget(ui->splitter);
//创建显示图表
QChart *chart = new QChart;
chart->setTitle("音频输入原始信号");
ui->chartView->setChart(chart);
lineSeries= new QLineSeries(); //序列
chart->addSeries(lineSeries);

QValueAxis *axisX = new QValueAxis; //坐标轴
axisX->setRange(0, displayPointsCount); //chart 显示 4000 个采样点数据
axisX->setLabelFormat("%g");
axisX->setTitleText("Samples");
QValueAxis *axisY = new QValueAxis; //坐标轴
axisY->setRange(0, 256); // UnsingedInt 采样, 数据范围 0-255
axisY->setTitleText("Audio level");

chart->setAxisX(axisX, lineSeries);
chart->setAxisY(axisY, lineSeries);
chart->legend()->hide();

ui->comboDevices->clear();
deviceList=QAudioDeviceInfo::availableDevices(QAudio::AudioInput);
for(int i=0;i<deviceList.count();i++)
{
    QAudioDeviceInfo device=deviceList.at(i);
    ui->comboDevices->addItem(device.deviceName());
}

if (deviceList.size()>0)
{
    ui->comboDevices->setCurrentIndex(0);
    curDevice =deviceList.at(0);//
}
else
{
    ui->actStart->setEnabled(false);
    ui->actDeviceTest->setEnabled(false);
    ui->groupBoxDevice->setTitle("支持的音频输入设置(无设备)");
}
}

```

构造函数创建了用于图表显示的 QChart 对象 chart, 创建了 QLineSeries 类型的序列 lineSeries, 创建了 X 和 Y 坐标轴; 其 X 轴的范围等于 0 到显示的数据点的总数 4000, Y 轴的范围是 0 至 256, 采用 8 位无符号整数, 采样数据范围是 0 至 255。

QAudioDeviceInfo::availableDevices(QAudio::AudioInput) 可以获取音频输入设备列表, 设备名称被添加到窗口上的 comboDevices 下拉列表框里。

### 3. 音频输入设备支持的格式

在主窗口的构造函数中, 向 comboDevices 下拉列表框中添加了系统所有的音频输入设备。在下拉列表框里选择一个设备时, 发射 currentIndexChanged(int index) 信号, 在其槽函数里获取设备支持的各种音频输入参数, 包括支持的音频编码、采样率、通道数、采样点类型和采样点大小等, 以此更新窗口上的组件显示。

```

void MainWindow::on_comboDevices_currentIndexChanged(int index)
{ //选择音频输入设备
    curDevice = deviceList.at(index); //当前音频设备
    ui->comboCodec->clear(); //支持的音频编码
    QStringList codecs = curDevice.supportedCodecs();
    for (int i = 0; i < codecs.size(); ++i)
        ui->comboCodec->addItem(codecs.at(i));

    ui->comboSampleRate->clear(); //支持的采样率
    QList<int> sampleRate = curDevice.supportedSampleRates();
    for (int i = 0; i < sampleRate.size(); ++i)
        ui->comboSampleRate->addItem(QString("%1").arg(sampleRate.at(i)));

    ui->comboChannels->clear(); //支持的通道数
    QList<int> Channels = curDevice.supportedChannelCounts();
    for (int i = 0; i < Channels.size(); ++i)
        ui->comboChannels->addItem(QString("%1").arg(Channels.at(i)));

    ui->comboSampleTypes->clear(); //支持的采样点类型
    QList<QAudioFormat::SampleType> sampleTypes = curDevice.supportedSampleTypes();
    for (int i = 0; i < sampleTypes.size(); ++i)
        ui->comboSampleTypes->addItem(SampleTypeString(sampleTypes.at(i)),
QVariant(sampleTypes.at(i)));

    ui->comboSampleSizes->clear(); //采样点大小
    QList<int> sampleSizes = curDevice.supportedSampleSizes();
    for (int i = 0; i < sampleSizes.size(); ++i)
        ui->comboSampleSizes->addItem(QString("%1").arg(sampleSizes.at(i)));

    ui->comboByteOrder->clear(); //字节序
    QList<QAudioFormat::Endian> endians = curDevice.supportedByteOrders();
    for (int i = 0; i < endians.size(); ++i)
        ui->comboByteOrder->addItem(ByteOrderString(endians.at(i)));
}

QString MainWindow::SampleTypeString(QAudioFormat::SampleType sampleType)
{ //将 QAudioFormat::SampleType 类型转换为字符串
    QString result("Unknown");
    switch (sampleType) {
        case QAudioFormat::SignedInt:
            result = "SignedInt";      break;
        case QAudioFormat::UnSignedInt:
            result = "UnSignedInt";    break;
        case QAudioFormat::Float:
            result = "Float";          break;
        case QAudioFormat::Unknown:
            result = "Unknown";
    }
    return result;
}

QString MainWindow::ByteOrderString(QAudioFormat::Endian endian)
{ //将 QAudioFormat::Endian 转换为字符串
    if (endian==QAudioFormat::LittleEndian)
        return "LittleEndian";
    else if (endian==QAudioFormat::BigEndian)

```

```

    return "BigEndian";
else
    return "Unknown";
}

```

创建一个 `QAudioInput` 对象时需要传递一个 `QAudioFormat` 类型作为参数, 用于指定音频输入配置, 而音频设备是否支持这些配置需要进行测试。窗口上的“测试音频设置”按钮可以进行测试, 代码如下:

```

void MainWindow::on_actDeviceTest_triggered()
{
    // 测试音频输入设备是否支持选择的设置
    QAudioFormat settings;
    settings.setCodec(ui->comboCodec->currentText());
    settings.setSampleRate(ui->comboSampleRate->currentText().toInt());
    settings.setChannelCount(ui->comboChannels->currentText().toInt());
    settings.setSampleType(QAudioFormat::SampleType(
        ui->comboSampleTypes->currentData().toInt()));
    settings.setSampleSize(ui->comboSampleSizes->currentText().toInt());
    if (ui->comboByteOrder->currentText()=="LittleEndian")
        settings.setByteOrder(QAudioFormat::LittleEndian);
    else
        settings.setByteOrder(QAudioFormat::BigEndian);

    if (curDevice.isFormatSupported(settings))
        QMessageBox::information(this, "音频测试", "测试成功, 输入设备支持此设置");
    else
        QMessageBox::critical(this, "音频测试", "测试失败, 输入设备不支持此设置");
}

```

`QAudioFormat` 类对象 `settings` 从界面上各个组件获取设置, 包括编码格式、采样率、通道数等, 然后用 `QAudioDeviceInfo::isFormatSupported()` 函数测试是否支持此设置; 如果不支持, 还可以使用 `QAudioDeviceInfo` 的 `nearestFormat()` 函数获取最接近的配置。

#### 4. 开始音频输入

单击窗口工具栏上“开始”按钮, 即可开始音频数据输入, 其代码如下:

```

void MainWindow::on_actStart_triggered()
{
    // 开始音频输入
    QAudioFormat defaultAudioFormat; // 缺省格式
    defaultAudioFormat.setSampleRate(8000);
    defaultAudioFormat.setChannelCount(1);
    defaultAudioFormat.setSampleSize(8);
    defaultAudioFormat.setCodec("audio/pcm");
    defaultAudioFormat.setByteOrder(QAudioFormat::LittleEndian);
    defaultAudioFormat.setSampleType(QAudioFormat::UnSignedInt);
    if (!curDevice.isFormatSupported(defaultAudioFormat))
    {
        QMessageBox::critical(this, "测试", "测试失败, 输入设备不支持此设置");
        return;
    }

    audioInput = new QAudioInput(curDevice, defaultAudioFormat, this);
    audioInput->setBufferSize(displayPointsCount);
    // 接收音频输入数据的流设备
    displayDevice = new QmyDisplayDevice(lineSeries, displayPointsCount, this);
}

```



```

connect(displayDevice,SIGNAL(updateBlockSize(qint64)),
        this,SLOT(on_IODevice_UpdateBlockSize(qint64)));
displayDevice->open(QIODevice::WriteOnly);

audioInput->start(displayDevice);
ui->actStart->setEnabled(false);
ui->actStop->setEnabled(true);
}

```

为了便于解析音频输入的原始数据，音频输入的配置采用固定的简单方式，而不是根据界面上的设置进行配置。音频输入配置固定为 8000 Hz 采样率、1 个通道、8 位无符号整数、audio/pcm 编码和小端字节序。

创建 QAudioInput 类对象 audioInput 时，传递 defaultAudioFormat 和 curDevice 作为参数，并设置缓冲区大小为 displayPointsCount（等于 4000）。

```

audioInput = new QAudioInput(curDevice,defaultAudioFormat, this);
audioInput->setBufferSize(displayPointsCount);

```

使用 setBufferSize 设置的缓冲区，需要在调用 start() 之前设置才有效。缓冲区的大小大于每次更新输入的原始数据块的大小，在图 15-4 中可见缓冲区大小为 4000，而每次更新的原始数据的数据字节数为 800。

随后程序创建一个 QmyDisplayDevice 类型的 IO 设备 displayDevice，这个类实现了流设备的 writeData() 函数，用于读取音频输入的数据并在曲线上显示。其构造函数接收 lineSeries 和 displayPointsCount 作为参数。再将 displayDevice 的信号 updateBlockSize() 与一个自定义槽函数关联，然后将 displayDevice 以只写方式打开。

最后调用 QAudioInput::start() 函数开始音频输入，以 displayDevice 作为 IO 流设备。

```
audioInput->start(displayDevice);
```

自定义槽函数 on\_IODevice\_UpdateBlockSize() 用于显示缓冲区大小和数据块大小，代码如下：

```

void MainWindow::on_IODevice_UpdateBlockSize(qint64 blockSize)
{ //显示缓冲区大小和数据块大小
    ui->LabBufferSize->setText(QString::asprintf(
        "QAudioInput::bufferSize()=%d",audioInput->bufferSize()));
    ui->LabBlockSize->setText(
        QString("IODevice 数据块字节数=%1").arg(blockSize));
}

```

## 5. 流设备 QmyDisplayDevice 的功能实现

使用 QAudioInput 获取音频输入数据时，需要使用一个 QIODevice 类型的设备作为流输出设备，一般采用 QFile 可以将数据记录到文件。本例中使用一个自定义的类 QmyDisplayDevice，用于获取音频输入数据并在曲线上实时显示。

QmyDisplayDevice 类的定义如下：

```

#include <QtCharts>
#include <QIODevice>
class QmyDisplayDevice : public QIODevice
{
    Q_OBJECT

```

```

public:
    explicit QmyDisplayDevice(QXYSeries * series, qint64 pointsCount, QObject *parent = 0);
protected:
    qint64 readData(char * data, qint64 maxSize);
    qint64 writeData(const char * data, qint64 maxSize);
private:
    QXYSeries *m_series;
    qint64 range=4000;
signals:
    void updateBlockSize(qint64 blockSize);
};

```

因为 QmyDisplayDevice 类从 QIODevice 继承，所以具有流数据读写功能。重新实现的 readData() 和 writeData() 是实现流数据读写功能的。

定义了 m\_series 和 range 两个私有变量。m\_series 是用于图表曲线显示的 QXYSeries 序列，range 缺省值为 4000，是序列最多显示的数据点数。构造函数里接收 series 和 pointsCount 对上述两个变量进行初始化。

QmyDisplayDevice 类的实现代码如下：

```

QmyDisplayDevice::QmyDisplayDevice(QXYSeries * series, qint64 pointsCount, QObject
*parent) : QIODevice(parent)
{ // 构造函数
    m_series= series;
    range=pointsCount;
}

qint64 QmyDisplayDevice::readData(char * data, qint64 maxSize)
{ // 流的读操作，不处理
    Q_UNUSED(data)
    Q_UNUSED(maxSize)
    return -1;
}

qint64 QmyDisplayDevice::writeData(const char * data, qint64 maxSize)
{ // 读取数据块内的数据，更新到序列
    QVector<QPointF> oldPoints = m_series->pointsVector();
    QVector<QPointF> points; //临时
    if (oldPoints.count() < range)
    { //m_series 序列的数据未满 4000 点，
        points = m_series->pointsVector();
    }
    else
    { //将原来 maxSize 至 4000 的数据点前移
        for (int i = maxSize; i < oldPoints.count(); i++)
            points.append(QPointF(i - maxSize, oldPoints.at(i).y()));
    }

    qint64 size = points.count();
    for (int k = 0; k < maxSize; k++) //数据块内的数据填充序列的尾部
        points.append(QPointF(k + size, (quint8)data[k]));

    m_series->replace(points);
    emit updateBlockSize(maxSize);
    return maxSize;
}

```

QmyDisplayDevice 无需实现流的读操作, 所以 readData()函数不做什么处理, 重点是流的写操作 writeData()函数的实现。

writeData()传递进来的参数 data 是数据块的指针, maxSize 是数据块的字节数, 这是需要读取出来的音频输入数据。

由于音频输入配置为 1 通道 8 位无符号的整数采样, 所以一个数据点就是一个字节的数据。

显示序列 m\_series 存储的显示数据点个数限定为 4000 个点, 大于 maxSize (此例中为 800), 所以对于序列的数据点的更新采用 FIFO (先入先出) 的方式。

更新临时数据点向量 points, 采用序列的 replace()函数替换序列原有的数据点向量, 是最快的方式。

writeData()函数最后发射信号 updateBlockSize(maxSize), 用于主窗口的关联槽函数 on\_IODevice\_UpdateBlockSize()则显示数据块的大小。

## 15.4 视频播放

使用 QMediaPlayer 可以进行视频文件解码, 视频播放必须将视频帧在某个界面组件上显示, 有 QVideoWidget 和 QGraphicsVideoItem 两种视频显示组件, 也可以从这两个类继承, 自定义视频显示组件。

QMediaPlayer 也可以结合 QMediaPlaylist 实现视频文件列表播放。

### 15.4.1 在 QVideoWidget 上播放视频

#### 1. 视频播放器实例程序

QVideoWidget 是用于显示视频的界面组件, 要在项目中使用 QVideoWidget, 需要在项目配置文件中添加下面一行语句:

```
Qt += multimediawidgets
```

使用 QMediaPlayer 和 QVideoWidget 实现的一个视频播放器实例程序 samp15\_5 运行时界面如图 15-5 所示。该程序没有使用 QMediaPlaylist, 只播放单个文件。

界面的主体部分是一个 QmyVideoWidget 类组件, 是从 QVideoWidget 继承的自定义视频显示组件, 重载了 mousePressEvent()事件, 鼠标单击可以暂停或继续播放; 重载了 keyPressEvent()事件, 在全屏状态下按 ESC 键可以退出全屏。原始的 QVideoWidget 类没有这些功能。

#### 2. 主窗口设计

主窗口是基于 QMainWindow 的类 MainWindow, 采用 UI 设计器设计界面。在设计视频显示



图 15-5 使用 QVideoWidget 的视频播放器

组件时，在窗体上放置一个 QWidget 组件，然后提升为 QmyVideoWidget 类。

主窗口功能主要是使用一个 QMediaPlayer 组件播放单个视频文件。媒体播放类 QMediaPlayer 的功能在 15.2 节介绍音频播放时已经详细介绍。

MainWindow 类定义如下（忽略了自动生成的界面组件的槽函数定义）：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QMediaPlayer *player;//视频播放器
    QString durationTime;
    QString positionTime;
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    //自定义槽函数
    void onStateChanged(QMediaPlayer::State state);
    void onDurationChanged(qint64 duration);
    void onPositionChanged(qint64 position);
private:
    Ui::MainWindow *ui;
};
```

MainWindow 类的功能实现代码如下，只播放单个文件，比 15.2 节的音频播放器功能简单，所以具体代码的功能这里不再赘述。

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    //构造函数
    ui->setupUi(this);
    player = new QMediaPlayer(this);//创建视频播放器
    player->setNotifyInterval(2000);//信息更新周期 2000 ms
    player->setVideoOutput(ui->videoWidget);//视频显示组件
    ui->videoWidget->setMediaPlayer(player);//设置显示组件的关联播放器
    connect(player, SIGNAL(stateChanged(QMediaPlayer::State)),
        this, SLOT(onStateChanged(QMediaPlayer::State)));
    connect(player, SIGNAL(positionChanged(qint64)),
        this, SLOT(onPositionChanged(qint64)));
    connect(player, SIGNAL(durationChanged(qint64)),
        this, SLOT(onDurationChanged(qint64)));
}

void MainWindow::onStateChanged(QMediaPlayer::State state)
{
    //播放器状态变化
    ui->btnPlay->setEnabled(!(state==QMediaPlayer::PlayingState));
    ui->btnPause->setEnabled(state==QMediaPlayer::PlayingState);
    ui->btnStop->setEnabled(state==QMediaPlayer::PlayingState);
}

void MainWindow::onDurationChanged(qint64 duration)
{
    //文件时长变化
    ui->sliderPosition->setMaximum(duration);
    int secs=duration/1000;//秒
    int mins=secs/60; //分钟
    secs=secs % 60;//余数秒
```



```

        durationTime=QString::asprintf("%d:%d",mins,secs);
        ui->LabRatio->setText(positionTime+"/"+durationTime);
    }
void MainWindow::onPositionChanged(qint64 position)
{ //文件播放位置变化
    if (ui->sliderPosition->isSliderDown())
        return; //如果正在拖动滑条, 退出
    ui->sliderPosition->setSliderPosition(position);
    int secs=position/1000; //秒
    int mins=secs/60; //分钟
    secs=secs % 60; //余数秒
    positionTime=QString::asprintf("%d:%d",mins,secs);
    ui->LabRatio->setText(positionTime+"/"+durationTime);
}
void MainWindow::on_btnAdd_clicked()
{ //打开文件
    QString curPath=QDir::homePath(); //获取系统当前目录
    QString dlgTitle="选择视频文件"; //对话框标题
    QString filter="wmv 文件 (*.wmv);;mp4 文件 (*.mp4);;所有文件 (*.*)";
    QString aFile=QFileDialog::getOpenFileName(this, dlgTitle, curPath, filter);
    if (aFile.isEmpty())
        return;
    QFileInfo fileInfo(aFile);
    ui->LabCurMedia->setText(fileInfo.fileName());
    player->setMedia(QUrl::fromLocalFile(aFile)); //设置播放文件
    player->play();
}
void MainWindow::on_btnPlay_clicked()
{ //播放
    player->play();
}
void MainWindow::on_btnPause_clicked()
{ //暂停
    player->pause();
}
void MainWindow::on_btnStop_clicked()
{ //停止
    player->stop();
}
void MainWindow::on_sliderVolumn_valueChanged(int value)
{ //调节音量
    player->setVolume(value);
}
void MainWindow::on_btnSound_clicked()
{ //静音按钮
    bool mute=player->isMuted();
    player->setMuted(!mute);
    if (mute)
        ui->btnSound->setIcon(QIcon(":/images/images/volumn.bmp"));
    else
        ui->btnSound->setIcon(QIcon(":/images/images/mute.bmp"));
}
void MainWindow::on_sliderPosition_valueChanged(int value)
{ //播放位置
    player->setPosition(value);
}

```

```

}
void MainWindow::on_pushButton_clicked()
{
    //全屏按钮
    ui->videoWidget->setFullScreen(true);
}

```

### 3. 自定义视频显示组件 QmyVideoWidget

原始的 QVideoWidget 在全屏后无法按 Esc 键退出全屏状态，为了实现按键和鼠标控制功能，从 QVideoWidget 继承了一个类 QmyVideoWidget，其定义如下：

```

class QmyVideoWidget : public QVideoWidget
{
private:
    QMediaPlayer *thePlayer;
protected:
    void keyPressEvent(QKeyEvent *event);
    void mousePressEvent(QMouseEvent *event);
public:
    QmyVideoWidget(QWidget *parent = Q_NULLPTR);
    void setMediaPlayer(QMediaPlayer *player);
};

```

私有变量 thePlayer 是关联的视频播放器，在开始播放之前由主窗口程序调用 setMediaPlayer() 进行设置，以便在 QmyVideoWidget 里对播放器进行控制。

下面是 QmyVideoWidget 类的实现代码：

```

void QmyVideoWidget::keyPressEvent(QKeyEvent *event)
{
    //按键事件，ESC 退出全屏状态
    if ((event->key() == Qt::Key_Escape)&&(isFullScreen()))
    {
        setFullScreen(false);
        event->accept();
        QVideoWidget::keyPressEvent(event);
    }
}

void QmyVideoWidget::mousePressEvent(QMouseEvent *event)
{
    //鼠标事件，单击控制暂停和继续播放
    if (event->button() == Qt::LeftButton)
    {
        if (thePlayer->state() == QMediaPlayer::PlayingState)
            thePlayer->pause();
        else
            thePlayer->play();
    }
    QVideoWidget::mousePressEvent(event);
}

void QmyVideoWidget::setMediaPlayer(QMediaPlayer *player)
{
    //设置播放器
    thePlayer = player;
}

```

## 15.4.2 在 QGraphicsVideoItem 上播放视频

QMediaPlayer 解码的视频还可以在 QGraphicsVideoItem 类组件上显示。QGraphicsVideoItem

是继承自 `QGraphicsItem` 的类，是适用于 `Graphics/View` 模式的图形显示组件。所以在使用 `QGraphicsVideoItem` 显示视频时，可以在显示场景中和其他图形组件组合显示，也可以使用 `QGraphicsItem` 类的放大、缩小、拖动、旋转等功能。

图 15-6 所示是使用 `QMediaPlayer` 和 `QGraphicsVideoItem` 播放视频的示例程序 `samp15_6` 运行时界面。窗口主体部分是一个 `QGraphicsView` 组件，在这个图形视图中创建了一个 `QGraphicsVideoItem` 对象用于显示播放的视频，还创建了两个图形组件，这些组件都可以拖动。



图 15-6 使用 `QGraphicsVideoItem` 的视频播放器

主窗口构造函数的部分代码如下：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    player = new QMediaPlayer(this);
    player->setNotifyInterval(2000);
    QGraphicsScene *scene = new QGraphicsScene(this);
    ui->graphicsView->setScene(scene);

    videoItem = new QGraphicsVideoItem;
    videoItem->setSize(QSizeF(360, 240));
    videoItem->setFlags(QGraphicsItem::ItemIsMovable
        | QGraphicsItem::ItemIsSelectable
        | QGraphicsItem::ItemIsFocusable);
    scene->addItem(videoItem);
    player->setVideoOutput(videoItem); // 设置视频显示图形组件

    connect(player, SIGNAL(stateChanged(QMediaPlayer::State)),
        this, SLOT(onStateChanged(QMediaPlayer::State)));
    connect(player, SIGNAL(positionChanged(qint64)),
        this, SLOT(onPositionChanged(qint64)));
    connect(player, SIGNAL(durationChanged(qint64)),
        this, SLOT(onDurationChanged(qint64)));
}
```

代码创建 `QGraphicsVideoItem` 类型变量 `videoItem`，并且设置其为可移动、可选择、可以获得

焦点。设置视频播放器 `player` 视频输出到 `videoItem` 上显示。

图 15-6 中程序其他功能的实现与实例 `samp15_5`，这里不再赘述。

## 15.5 摄像头的使用

### 15.5.1 摄像头控制概述

Qt 多媒体模块为摄像头控制提供了几个类，可以用于获取摄像头设备信息，通过摄像头进行拍照和录像。

(1) 摄像头设备信息类 `QCameraInfo`。

`QCameraInfo` 用于获取系统的摄像头设备信息，有两个静态函数获取摄像头设备：

- `QList<QCameraInfo> availableCameras()`，返回 `QCameraInfo` 类的列表，表示系统可用的摄像头设备列表；
- `QCameraInfo defaultCamera()`，返回系统缺省的摄像头设备信息。

`QCameraInfo` 有几个函数表示摄像头信息：

- `QString description()`，返回摄像头设备描述；
- `QString deviceName()`，返回摄像头设备名称；
- `QCamera::Position position()`，返回摄像头的位置，如手机设备上一般有两个摄像头，前置摄像头位置类型为 `QCamera::FrontFace`，后置摄像头位置类型为 `QCamera::BackFace`，未指定位置的是 `QCamera::UnspecifiedPosition`。

(2) 摄像头控制类 `QCamera`。

`QCamera` 是用于控制摄像头的类，创建 `QCamera` 对象时需传递一个 `QCameraInfo` 对象作为参数，`QCamera` 主要的功能函数包括以下几个。

- `setViewfinder()`，为摄像头指定一个 `QVideoWidget` 或 `QGraphicsVideoItem` 对象作为取景器，用于摄像头图像预览。
- `QCameraExposure *exposure()`，返回用于曝光控制的对象。
- `QCameraFocus *focus()`，返回用于聚焦控制的对象。
- `setCaptureMode(QCamera::CaptureModes mode)`，用于设置摄像头处于不同的工作模式，`QCamera::CaptureModes` 枚举类型的取值有：

`QCamera::CaptureViewfinder`，取景器模式；

`QCamera::CaptureStillImage`，抓取静态图片模式；

`QCamera::CaptureVideo`，视频录制模式。

- `bool isCaptureModeSupported(CaptureModes mode)`，判断摄像头是否支持某种抓取模式。

(3) 静态图片抓取类 `QCameraImageCapture`。

`QCameraImageCapture` 用于控制摄像头进行静态图片的抓取。

(4) 视频和音频录制类 `QMediaRecorder`。



QMediaRecorder 通过摄像头和音频输入设备进行录像。

**注意** 使用 Qt 多媒体模块的摄像头相关类无法在 Windows 平台上进行视频录制，只能进行静态图片抓取，但是在 Linux 平台上可以实现静态图片抓取和视频录制。所以，本节的实例程序在 Linux 平台（64 位版本的 Ubuntu 16.04）上测试。

Qt 多媒体模块的功能实现是依赖于平台的。在 Windows 平台上，Qt 多媒体模块依赖于两个插件，一个是使用 Microsoft DirectShow API 的插件，DirectShow API 在 Windows 98 引入，在 Windows XP 以后就逐渐过时了；另一个是 Windows Media Foundation (WMF) 架构的插件，WMF 插件在 Windows Vista 引入，用于替代 DirectShow API。

Qt 中的 WMF 插件目前无法提供摄像头支持，对摄像头的有限支持是由 DirectShow 插件提供的，目前只能显示取景器和抓取静态图片，其他大部分功能不支持。所以，目前在 Windows 平台上，Qt 的摄像头控制不支持视频录制功能，也不支持底层的视频功能，如使用 QVideoProbe 监测视频帧。

对于这些限制的原文介绍可以参考 Qt 官网上“Qt Multimedia on Windows”的解释。

所以，本节的实例程序 samp15\_7 在 64 位版本的 Ubuntu 16.04 桌面系统上测试，Ubuntu 系统运行于 VirtualBox 虚拟机上。图 15-7 所示是程序运行时界面。



图 15-7 实例 samp15\_7 运行时界面

工具栏上几个按钮的功能一目了然，“摄像头参数”里显示摄像头的参数，以及录制视频的一些设置。窗口工作区左侧的“摄像头预览”框里显示摄像头的实时图像预览，右侧“抓取图片”框里显示抓取的静态图片。

## 15.5.2 实例主窗口设计与初始化

实例 samp15\_7 的主窗口是基于 QMainWindow 的类 MainWindow，界面由 UI 设计器设计。在窗口左侧“摄像头预览”框里放置一个 QWidget 组件，并提升为 QCameraViewfinder，这是从 QVideoWidget 继承的类，专门用于摄像头预览显示。在窗口右侧“抓取图片”框里放置一个 QLabel 组件，用于显示抓取的图片。

窗口的工具栏按钮由 Action 生成，设计的 Action 列表如图 15-8 所示。







Name	Used	Text	Shortcut	Checkable	ToolTip
 actStartCamera	<input checked="" type="checkbox"/>	开启摄像头		<input type="checkbox"/>	开启摄像头
 actStopCamera	<input checked="" type="checkbox"/>	关闭摄像头		<input type="checkbox"/>	关闭摄像头
 actVideoRecord	<input checked="" type="checkbox"/>	开始录像		<input type="checkbox"/>	开始录像
 actCapture	<input checked="" type="checkbox"/>	抓图		<input type="checkbox"/>	抓图
 actQuit	<input checked="" type="checkbox"/>	退出		<input type="checkbox"/>	退出
 actVideoStop	<input checked="" type="checkbox"/>	停止录像		<input type="checkbox"/>	停止录像

图 15-8 实例 samp15\_7 设计的 Action

主窗口类 MainWindow 的定义如下（省略了自动生成的界面组件的槽函数定义）：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
private:
    QCamera      *curCamera=Q_NULLPTR;
    QCameraImageCapture *imageCapture; //抓图
    QMediaRecorder *mediaRecorder; //录像
    QLabel      *LabCameraState;
    QLabel      *LabInfo;
    QLabel      *LabCameraMode;
    void    iniCamera(); //初始化
    void    iniImageCapture(); //初始化静态抓图
    void    iniVideoRecorder(); //初始化视频录制
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    //QCamera 的槽函数
    void    on_cameraStateChanged(QCamera::State state);
    void    on_cameraCaptureModeChanged(QCamera::CaptureModes mode);
    //QCameraImageCapture 的槽函数
    void    on_imageReadyForCapture(bool ready);
    void    on_imageCaptured(int id, const QImage &preview);
    void    on_imageSaved(int id, const QString &fileName);
    // QMediaRecorder 的槽函数
    void    on_videoStateChanged(QMediaRecorder::State state);
    void    on_videoDurationChanged(qint64 duration);
private:
    Ui::MainWindow *ui;
};
```

这里定义了 QCamera、QCameraImageCapture 和 QmediaRecorder 3 个类型的私有变量。定义了 3 个私有函数，分别用于 3 个对象的创建和初始化。

定义了多个槽函数，分别与3个对象的相应信号关联。

下面是 MainWindow 的构造函数的代码：

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setUpUi(this);
    LabCameraState=new QLabel("摄像头 state: ");
    LabCameraState->setMinimumWidth(150);
    ui->statusBar->addWidget(LabCameraState);
    LabCameraMode=new QLabel("抓取模式: ");
    LabCameraMode->setMinimumWidth(150);
    ui->statusBar->addWidget(LabCameraMode);
    LabInfo=new QLabel("");
    ui->statusBar->addPermanentWidget(LabInfo);

    QList<QCameraInfo> cameras = QCameraInfo::availableCameras();
    if (cameras.size()>0)
    {
        iniCamera(); //初始化摄像头
        iniImageCapture(); //初始化静态抓图
        iniVideoRecorder(); //初始化视频录制
        curCamera->start();
    }
}
```

构造函数首先要创建状态栏上显示信息的标签，然后使用 QCameraInfo 的静态函数 availableCameras() 获取摄像头列表 cameras，如果存在摄像头设备，就依次调用3个初始化函数进行初始化，然后打开摄像头。

### 15.5.3 QCamera 对象创建与控制

iniCamera() 函数进行 QCamera 对象的创建与初始化，在 MainWindow 的构造函数里调用，下面是 iniCamera() 函数的代码：

```
void MainWindow::iniCamera()
{ // 创建 QCamera 对象
    QCameraInfo curCameraInfo=QCameraInfo::defaultCamera(); //获取缺省摄像头
    ui->comboCamera->addItem(curCameraInfo.description());
    ui->comboCamera->setCurrentIndex(0);

    curCamera=new QCamera(curCameraInfo, this); //创建摄像头对象
    QCameraViewfinderSettings viewfinderSettings;
    viewfinderSettings.setResolution(640, 480);
    viewfinderSettings.setMinimumFrameRate(15.0);
    viewfinderSettings.setMaximumFrameRate(30.0);
    curCamera->setViewfinderSettings(viewfinderSettings);
    curCamera->setViewfinder(ui->viewFinder); //设置预览框
    curCamera->setCaptureMode(QCamera::CaptureViewfinder);
    //判断摄像头是否支持抓图、录制视频
    ui->checkStillImage->setChecked(
        curCamera->isCaptureModeSupported(QCamera::CaptureStillImage)); //抓图
    ui->checkVideo->setChecked(
```

```

    curCamera->isCaptureModeSupported(QCamera::CaptureVideo)); // 视频录制

    connect(curCamera, SIGNAL(stateChanged(QCamera::State)),
            this, SLOT(on_cameraStateChanged(QCamera::State)));
    // Windows 平台上不支持 captureModeChanged() 信号
    connect(curCamera, SIGNAL(captureModeChanged(QCamera::CaptureModes)),
            this, SLOT(on_cameraCaptureModeChanged(QCamera::CaptureModes)));
}

```

这里用 `QCameraInfo::defaultCamera()` 获取缺省的摄像头设备，然后用于创建 `QCamera` 对象实例 `curCamera`。

`QCamera::setViewfinder()` 函数用于设置取景框预览组件。

`QCamera::setCaptureMode()` 函数设置摄像头的抓取模式。

然后查询 `curCamera` 是否支持抓图、支持录像，并更新界面上的复选框的状态。

为 `curCamera` 的两个信号分别设置了关联的槽函数。`stateChanged()` 在摄像头状态变化时发射，`captureModeChanged()` 则在摄像头抓取模式变化时发射，不过在 Windows 平台上不会发射 `captureModeChanged()` 信号。这两个槽函数的代码如下：

```

void MainWindow::on_cameraStateChanged(QCamera::State state)
{
    switch (state)
    {
        case QCamera::UnloadedState:
            LabCameraState->setText("摄像头 state: UnloadedState");    break;
        case QCamera::LoadedState:
            LabCameraState->setText("摄像头 state: LoadedState");        break;
        case QCamera::ActiveState:
            LabCameraState->setText("摄像头 state: ActiveState");
    }
    ui->actStartCamera->setEnabled(state!=QCamera::ActiveState);
    ui->actStopCamera->setEnabled(state==QCamera::ActiveState);
}

void MainWindow::on_cameraCaptureModeChanged(QCamera::CaptureModes mode)
{
    if (mode==QCamera::CaptureStillImage)
        LabCameraMode->setText("抓取模式: StillImage");
    else if (mode==QCamera::CaptureVideo)
        LabCameraMode->setText("抓取模式: Video");
    else
        LabCameraMode->setText("抓取模式: Viewfinder");
}

```

窗口工具栏上的“开启摄像头”和“关闭摄像头”两个按钮控制摄像头的开启和关闭，其关联 Action 的槽函数代码如下：

```

void MainWindow::on_actStartCamera_triggered()
{ // 开启摄像头
    curCamera->start();
}

void MainWindow::on_actStopCamera_triggered()
{ // 关闭摄像头
    curCamera->stop();
}

```



开启和关闭摄像头时，QCamera 对象会发射 stateChanged()信号。开启摄像头之后，摄像头状态为 QCamera::ActiveState，关闭之后状态会变成 QCamera::LoadedState。

### 15.5.4 QCameraImageCapture 抓取静态图片

QCameraImageCapture 类对象 imageCapture 用于通过摄像头抓取静态图片，iniImageCapture() 函数用于创建 imageCapture 并进行初始化设置，MainWindow 构造函数调用此函数。IniImageCapture()函数的代码如下：

```
void MainWindow::iniImageCapture()
{ //创建 QCameraImageCapture 对象
    imageCapture = new QCameraImageCapture(curCamera, this);
    imageCapture->setBufferFormat(QVideoFrame::Format_Jpeg); //缓冲区格式
    imageCapture->setCaptureDestination(
        QCameraImageCapture::CaptureToFile); //保存目标
    connect(imageCapture, SIGNAL(readyForCaptureChanged(bool)),
        this, SLOT(on_imageReadyForCapture(bool)));
    connect(imageCapture, SIGNAL(imageCaptured(int, const QImage &)),
        this, SLOT(on_imageCaptured(int, const QImage &)));
    connect(imageCapture, SIGNAL(imageSaved(int, const QString &)),
        this, SLOT(on_imageSaved(int, const QString &)));
}
```

创建 QCameraImageCapture 对象 imageCapture 时传递 QCamera 对象 curCamera 作为输入参数，建立与摄像头设备的关联。

setBufferFormat(QVideoFrame::Format\_Jpeg)设置缓冲区里图片为 JPG 格式。

setCaptureDestination(QCameraImageCapture::CaptureToFile)设置抓图存储目标为文件，抓取的图片文件会自动保存到用户目录的“图片”文件夹里。

为 imageCapture 的 3 个信号关联了自定义槽函数，这 3 个槽函数的代码如下：

```
void MainWindow::on_imageReadyForCapture(bool ready)
{ //可以抓图了
    ui->actCapture->setEnabled(ready);
}

void MainWindow::on_imageCaptured(int id, const QImage &preview)
{ //抓取图片后显示
    Q_UNUSED(id);
    QImage scaledImage = preview.scaled(ui->LabCapturedImage->size(),
        Qt::KeepAspectRatio, Qt::SmoothTransformation);
    ui->LabCapturedImage->setPixmap(QPixmap::fromImage(scaledImage));
}

void MainWindow::on_imageSaved(int id, const QString &fileName)
{ //文件保存后显示保存的文件名
    Q_UNUSED(id);
    LabInfo->setText("图片保存为: "+fileName);
}
```

单击窗口工具栏上的“抓图”按钮可实现抓取静态图片功能，其代码如下：

```
void MainWindow::on_actCapture_triggered()
{ //抓图 按钮
    if (curCamera->captureMode() != QCamera::CaptureStillImage)
```

```

        curCamera->setCaptureMode(QCamera::CaptureStillImage);
        imageCapture->capture();
    }

```

这里首先将 `curCamera` 的抓取模式设置为 `QCamera::CaptureStillImage`，然后使用 `QCameraImageCapture::capture()` 函数抓图。

## 15.5.5 QMediaRecorder 视频录制

`QMediaRecorder` 类对象 `mediaRecorder` 用于通过摄像头进行视频录制，`iniVideoRecorder()` 函数创建 `mediaRecorder` 并进行初始化设置，`MainWindow` 构造函数调用此函数。`iniVideoRecorder()` 函数代码如下：

```

void MainWindow::iniVideoRecorder()
{//创建 QMediaRecorder 对象
    mediaRecorder = new QMediaRecorder(curCamera, this);
    ui->chkMute->setChecked(mediaRecorder->isMuted());
    connect(mediaRecorder, SIGNAL(stateChanged(QMediaRecorder::State)),
            this, SLOT(on_videoStateChanged(QMediaRecorder::State)));
    connect(mediaRecorder, SIGNAL(durationChanged(qint64)),
            this, SLOT(on_videoDurationChanged(qint64)));
}

```

`mediaRecorder` 创建时传递 `QCamera` 对象 `curCamera` 作为输入参数。

为 `mediaRecorder` 的两个信号设置了关联槽函数，两个槽函数的代码如下：

```

void MainWindow::on_video_stateChanged(QMediaRecorder::State state)
{//状态变化
    ui->actVideoRecord->setEnabled(state!=QMediaRecorder::RecordingState);
    ui->actVideoStop->setEnabled(state==QMediaRecorder::RecordingState);
}
void MainWindow::on_video_durationChanged(qint64 duration)
{
    ui->LabDuration->setText(QString("录制时间:%1 秒").arg(duration/1000));
}

```

`on_videoStateChanged()` 函数会根据当前状态控制界面按钮的使能状态，`on_videoDurationChanged()` 函数用于录制视频时显示已录制的持续时间。

开始录像之前，需要单击“视频保存文件”按钮设置一个视频录制保存的文件。“开始录像”和“停止录像”两个按钮的代码如下：

```

void MainWindow::on_actVideoRecord_triggered()
{//开始录像
    if (!mediaRecorder->isAvailable())
    {
        QMessageBox::critical(this, "错误",
                               "不支持视频录制! \n mediaRecorder->isAvailable()==false");
        return;
    }
    if (curCamera->captureMode()!=QCamera::CaptureVideo)
        curCamera->setCaptureMode(QCamera::CaptureVideo);

    QString selectedFile=ui->editOutputFile->text().trimmed();

```

```
if (selectedFile.isEmpty())
{
    QMessageBox::critical(this, "错误", "请先设置录音输出文件");
    return;
}
if (QFile::exists(selectedFile))
    if (!QFile::remove(selectedFile))
    {
        QMessageBox::critical(this, "错误", "所设置录音输出文件被占用，无法删除");
        return;
    }

mediaRecorder->setOutputLocation(QUrl::fromLocalFile(selectedFile));
mediaRecorder->record();
}

void MainWindow::on_actVideoStop_triggered()
{ // 停止录像
    mediaRecorder->stop();
}
```

开始录像之前，还需要将摄像头的抓取状态设置为 `QCamera::CaptureVideo`，然后检查设置的视频输出文件，当文件没问题后用 `setOutputLocation()` 函数设置视频录制输出文件。

调用 `QMediaRecorder` 的 `record()` 函数开始录像，`stop()` 函数停止录像。

# 应用程序设计辅助功能

本章介绍 Qt 应用程序设计的一些辅助功能，包括设计多语言界面、使用样式表定制界面和组件的外观、使用 QStyle 设置界面外观，以及应用程序发布等。这些辅助功能不一定在每个应用程序设计里都会用到，但也是非常有用的一些功能，如在大型的软件设计中一般要考虑使用多语言界面，完成应用程序开发后还要考虑程序发布的问题。

## 16.1 多语言界面

### 16.1.1 多语言界面设计概述

有些软件需要开发多语言界面版本，如中文版和英文版，并且在软件里可以方便地切换界面语言。Qt 为多语言界面提供了很好的支持，使用 Qt 的一些规则和工具，可以很方便地为应用程序开发提供多语言界面支持。

用 Qt 开发多语言界面应用程序，主要包括以下几个步骤。

(1) 在程序设计阶段，程序代码中每一个用户可见的字符串都用 `tr()` 函数封装，以便 Qt 提取界面字符串用于生成翻译资源文件。用 UI 设计器可视化设计窗体时统一用一种语言，如汉语。

(2) 在项目配置文件（.pro 文件）中设置需要导出的翻译文件（.ts 文件）名称，使用 `lupdate` 工具扫描项目文件中需要翻译的字符串，并生成翻译文件。

(3) 使用 Qt 的 `Linguist` 程序打开生成的翻译文件，将程序中的字符串翻译为需要的语言，如将所有中文字符串翻译为英文。

(4) 使用 `lrelease` 工具编译翻译好的翻译文件，生成更为紧凑的“.qm”文件。

(5) 在应用程序中用 `QTranslator` 调用不同的“.qm”文件，实现不同的语言界面。

### 16.1.2 tr()函数的使用

为了让 Qt 能自动提取程序中用户可见的字符串，对于每个字符串都需要使用 `tr()` 函数封装。`tr()` 是 `QObject` 的一个静态函数，在使用了 `Q_OBJECT` 宏定义的类或 `QObject` 的子类中，都可以直接使用 `tr()` 函数，否则需要使用 `QObject::tr()` 进行调用。或者在类定义中用 `Q_DECLARE_TR_FUNCTIONS` 宏把 `tr()` 函数添加到类中之后，再直接调用 `tr()` 函数。

`tr()` 函数的定义是：



```
QString QObject::tr(const char *sourceText, const char *disambiguation = Q_NULLPTR,
int n = -1)
```

其中, sourceText 是源字符串, disambiguation 是为翻译者提供额外信息的字符串, 用于对一些容易混淆的地方作说明, 内容如下:

```
LabCellPos = new QLabel(tr("当前单元格: "), this);
QMessageBox::information(this, tr("信息"), tr("信息提示? "),
    QMessageBox::Yes);
QString str1=tr("左右", "大约的意思");
QString str2=tr("左右", "掌握、控制的意思");
```

使用 tr() 函数, 需要注意以下一些事项。

- 尽量使用常量字符串, 不要使用字符串变量。在 tr() 函数中应直接传递字符串常量, 而不是用变量传递字符串, 如下面的代码使用了字符串变量, 使用 lupdate 工具提取项目中的字符串时, 将不能提取“不能删除记录”这个字符串。

```
char *errorStr="不能删除记录";
QString str2=tr(errorStr);
```

- 使用字符串变量时需要用 Qt\_TR\_NOOP() 宏进行标记。若要在 tr() 函数中使用字符串变量, 需要在定义字符串的地方用 Qt\_TR\_NOOP() 宏进行标记, 这在使用字符串数组时比较有用, 例如:

```
const char *cities[4]={Qt_TR_NOOP("Beijing"),
    Qt_TR_NOOP("Shanghai"),
    Qt_TR_NOOP("Qingdao"),
    Qt_TR_NOOP("Wuhan") };
for (int i=0; i<4; i++)
    comboBox->addItem(tr(cities[i]));
```

- tr() 不能使用拼接的动态字符串。tr() 不能使用拼接的动态字符串, 例如, 下面的用法是错误的:

```
LabCellPos->setText(tr("第"+ QString::number(current.row()) +"行");
```

正确的方式如下。

```
LabCellPos->setText(tr("第 %1 行").arg(current.row()));
```

翻译的字符串是“第 %1 行”, 然后再用 QString 的 arg() 去替换占位符“%1”的内容。

- Qt\_NO\_CAST\_FROM\_ASCII 的作用。在一个需要翻译为多语言的应用程序中, 如果编写程序时忘了对某个字符串使用 tr() 函数, lupdate 生成的翻译资源文件就会遗漏这个字符串。为了避免这种疏忽错误, 可以在项目配置文件 (.pro 文件) 中添加如下的定义:

```
DEFINES += Qt_NO_CAST_FROM_ASCII
```

这样在编译时, 会禁止从 const char\* 到 QString 的隐式转换, 强制每个字符串都必须使用 tr() 或 QLatin1String() 封装, 避免出现遗漏未翻译的字符串。

### 16.1.3 生成语言翻译文件

要生成多语言界面相关的翻译文件, 除了之前所说的在对每个字符串都使用 tr() 函数封装之

外，还需要在项目配置文件（.pro 文件）中使用 TRANSLATIONS 定义语言翻译文件（.ts 文件），并使用 lupdate 工具生成语言翻译文件。

作为实例，将第 6 章的 samp6\_2 作为多语言界面设计实例，复制项目 samp6\_2 的全部文件，并将项目名称更改为 samp16\_1。在项目的配置文件中增加如下的设置语句：

```
TRANSLATIONS +=samp16_1_cn.ts\
               samp16_1_en.ts
```

这里设置生成两个语言翻译文件“samp16\_1\_cn.ts”和“samp16\_1\_en.ts”，分别是中文和英文翻译文件。文件名称可以任意设计，只要有所区分即可。

原始的程序设计采用中文。为了便于进行语言切换，在主窗口上设置了两个工具栏按钮，分别用于切换中文和英文界面（如图 16-1 所示），界面语言切换的代码实现会在下一小节介绍。

为了让 lupdate 工具能提取项目代码内的字符串，这里对项目程序里的字符串全部采用 tr() 函数封装，对于不符合 tr() 函数使用规则的地方进行修改。例如，之前在状态栏上显示当前单元格的行号和列号的程序是：

```
LabCellPos->setText(QString::asprintf("当前单元格: %d 行, %d 列",
                                       current.row(),current.column()));
```

现在就修改为如下形式：

```
LabCellPos->setText(tr("当前单元格: %1 行, %2 列"
                      ).arg(current.row()).arg(current.column()));
```

在项目设计期间，任何时候都可以使用 lupdate 工具生成或更新翻译文件，方法是单击 Qt Creator 主菜单的“Tools”→“External”→“Qt 语言家”→“Update Translations(lupdate)”菜单项，若项目的源程序目录下没有 samp16\_1\_cn.ts 和 samp16\_1\_en.ts 这两个文件，就会自动生成，如果文件已经存在，则会更新这两个文件的内容。

#### 16.1.4 使用 Qt Linguist 翻译 ts 文件

生成的 samp16\_1\_cn.ts 和 samp16\_1\_en.ts 文件内包含了项目源程序和 UI 界面里的所有字符串，使用 Qt Linguist 可以将这些字符串翻译为需要的语言版本。在 Qt 安装后的程序组里可以找到 Qt Linguist 软件。

samp16\_1\_cn.ts 是中文界面的翻译文件，由于源程序的界面就是用中文设计的，所以无需再翻译。samp16\_1\_en.ts 是英文翻译文件，需要将提取的所有中文字符串翻译为英文。

在 Linguist 软件中打开文件 samp16\_1\_en.ts，当第一次打开一个 ts 文件时，Linguist 会出现如图 16-2 所示的语言设置对话框，用于设置目标语言和所在国家和地区。这个对话框也可以通过 Linguist 主菜单的“编辑”→“翻译文件设置”菜单项调出。samp16\_1\_en.ts 是用于英文界面的翻



图 16-1 实例 samp16\_1 运行时主窗口

译文件，所以选择语言“English”，国家/地区可选择“UnitedStates”。

打开 samp16\_1\_en.ts 文件后的 Linguist 软件界面如图 16-3 所示。左侧“上下文”列表里列出了项目中的所有窗口或类，这个项目有 4 个窗口。“字符串”列表里列出了从项目的 UI 窗口和代码文件中提取的字符串，右侧“短语和表单”会显示窗口界面的预览或字符串在源程序中出现的代码段。

在“字符串”列表中选择一个源文后，在下方会出现译文编辑框，在此填写字符串对应的英文译文。Linguist 可以同时打开项目的多个 ts 文件，在选中一个源文后，在下方会出现对应的多个语言的译文编辑框，可以同时翻译为多个语言版本。

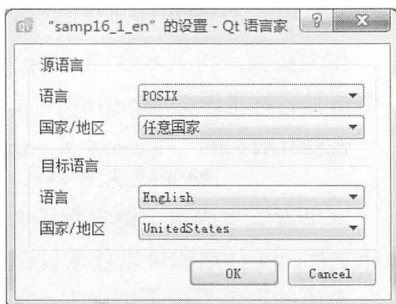


图 16-2 Linguist 软件设置语种的对话框

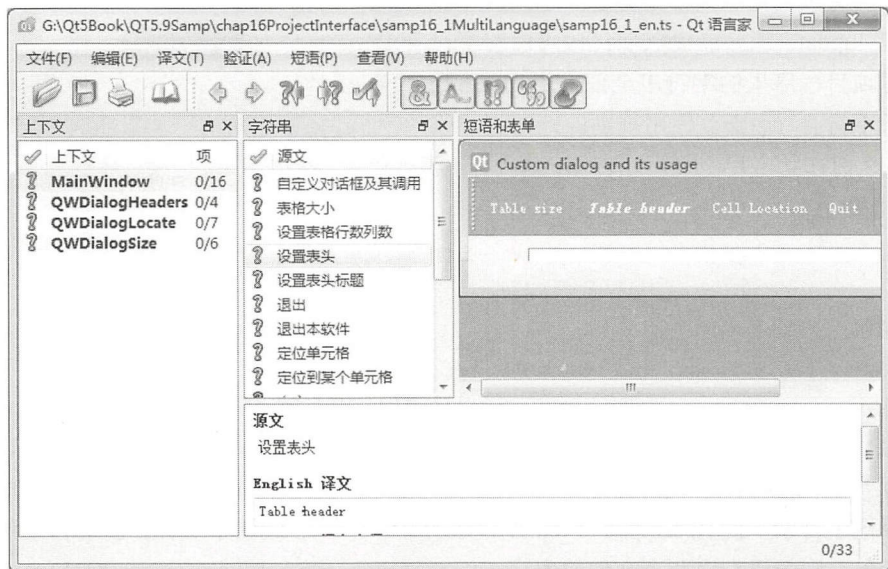


图 16-3 Linguist 软件翻译 ts 文件的界面

### 16.1.5 调用翻译文件改变界面语言

#### 1. 生成 qm 文件

使用 Linguist 软件编辑翻译文件，将所有字符串都翻译后，在 Qt Creator 中单击主菜单项“Tools”→“External”→“Qt 语言家”→“Release Translations(lrelease)”，会在项目源程序目录下生成与 ts 文件对应的 qm 文件，这是更为紧凑的翻译文件。本实例生成的是 samp16\_1\_cn.qm 和 samp16\_1\_en.qm。

#### 2. 项目启动时设置界面语言

使用 QTranslator 类设置界面的不同语言版本，需在应用程序启动时设置界面语言翻译文件，即在 main() 函数中进行处理。项目 samp16\_1 的 main.cpp 的代码如下：



```

#include "mainwindow.h"
#include <QApplication>
#include <QTranslator>
#include <QSettings>
QTranslator *trans=NULL;
QString readSetting();

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    trans=new QTranslator;
    QString curLang=readSetting(); //读取语言设置
    if (curLang=="EN")
        trans->load("samp16_1_en.qm");
    else
        trans->load("samp16_1_cn.qm");
    app.installTranslator(trans);
    MainWindow w;
    w.show();
    return app.exec();
}

QString readSetting()
{ //从注册表读取上次设置的语言
    QString organization="WWB-Qt";
    QString appName="samp16_1";
    QSettings settings(organization,appName);
    QString Language=settings.value("Language","EN").toString();
    return Language;
}

```

这里定义了一个全局变量 `trans`，在 `main.cpp` 里定义了一个函数 `readSetting()`，用于从注册表里读取上次设置的界面语言版本。注册表里数据的读取和写入使用到 `QSettings` 类，在 6.5 节有详细介绍。

在 `main()` 函数中，创建 `QTranslator` 类的对象 `trans` 后，调用 `readSetting()` 函数从注册表里读取上次的语言版本，若为“EN”，就用 `load()` 函数载入编译后的英文翻译文件 `samp16_1_en.qm`，否则就载入中文翻译文件 `samp16_1_cn.qm`。然后再执行 `app.installTranslator(trans)`，就可以给应用程序安装翻译器，实现需要的界面版本。

`samp16_1_en.qm` 和 `samp16_1_cn.qm` 文件是被编译到项目的可执行文件中的，无需将这两个文件复制到可执行文件目录下。

在 `main()` 函数里加载翻译器是相比之下一劳永逸的方法，这样一来，随后应用程序的任何窗口都会自动应用开始设置的语言。所以，一些大型的软件在重新设置了语言版本后，通常都会要求重新启动软件才生效。

### 3. 动态切换语言

在软件运行时可以动态切换语言，即无需重启软件就可以切换界面语言。在 `samp16_1` 的主窗口上有“中文”和“English”两个工具栏按钮，用于实现中文和英文界面的切换。下面是这两个按钮的响应代码：

```

void MainWindow::on_actLang_CN_triggered()
{ //中文界面

```



```

    qApp->removeTranslator(trans);
    delete trans;
    trans=new QTranslator;
    trans->load("sample_1_cn.qm");
    qApp->installTranslator(trans);
    ui->retranslateUi(this); //刷新界面字符串
    QSettings settings("WWB-Qt","sample_1");
    settings.setValue("Language","CN");
}

void MainWindow::on_actLang_EN_triggered()
{
    //英文界面
    qApp->removeTranslator(trans);
    delete trans;
    trans=new QTranslator;
    trans->load("sample_1_en.qm");
    qApp->installTranslator(trans);
    ui->retranslateUi(this); //刷新界面字符串
    QSettings settings("WWB-Qt","sample_1");
    settings.setValue("Language","EN");
}

```

一个应用程序只能加载一个翻译器，因为在 main()函数里已经加载了一个翻译器，所以需要先移除当前的翻译器，才能重新创建新的翻译器，加载翻译文件，并为应用程序重新加载新翻译器。

完成这些后还必须调用 UI 的 retranslateUi()函数来刷新界面。

retranslateUi()函数是在窗口的“ui\_”头文件中自动生成的，如 mainwindow.ui 主窗口对应的头文件 ui\_mainwindow.h 里就有函数 retranslateUi()，这个函数使用 QApplication::translate()函数将所有界面字符串进行翻译，类似于 tr()函数的功能。

窗口在被创建时会自动调用此 retranslateUi()函数，在程序运行中动态切换界面语言时，必须手工调用此 retranslateUi()函数，才可以立即更新界面语言。

若不是用 UI Designer 设计的窗口，而是完全由代码实现的窗口界面，需要专门设计一个 retranslateUi()函数，将所有界面字符串用 tr()函数进行翻译，并在动态切换语言时调用此函数。很显然，这样做比较麻烦，特别是当软件比较大，窗口非常多时。所以，大型的软件在重新设置语言版本后，一般要求重新启动软件才生效，在程序启动时根据上次的设置加载一次翻译器即可。

按钮响应代码的最后是将设置的语言版本写入注册表，以便下次程序启动时自动加载相应的语言。

## 16.2 使用样式表自定义界面

### 16.2.1 Qt 样式表

Qt 样式表 (style sheet) 是用于定制用户界面的强有力的机制，其概念、术语是受到 HTML 中的级联样式表 (Cascading Style Sheets, CSS) 启发而来的，只是 Qt 样式表是应用于窗体界面的。

与 HTML 的 CSS 类似，Qt 的样式表是纯文本的格式定义，在应用程序运行时可以载入和解

析这些样式定义。使用样式表可以定义各种界面组件（QWidget 类及其子类）的样式，从而使应用程序的界面呈现不同的效果。很多软件具有换肤功能，使用 Qt 的样式表就可以容易地实现这样的功能。

在 UI 设计器中集成了 Qt 样式表的编辑功能。在设计窗时，选择窗体或某个界面组件，单击鼠标右键，在弹出的快捷菜单中选择“Change styleSheet...”菜单项就可以出现样式表编辑对话框。图 16-4 所示是某个窗体的样式表编辑对话框，已经对窗体和一些类设置了样式定义，例如：

```
QWidget{
    background-color: rgb(255, 255, 0);
    color: rgb(255, 0, 0);
    font: 12pt "宋体";
}
```

这定义了 QWidget 类的背景颜色、字体大小和名称、前景颜色。这个样式定义会应用于 QWidget 类及其子类。

```
QLineEdit{
    border: 2px groove gray;
    border-radius: 5px;
    padding: 2px 4px;
    border-color: rgb(12, 45, 68);
}
```

这定义了 QLineEdit 类的显示效果，包括边框宽度、圆角边框的半径和边框颜色等。

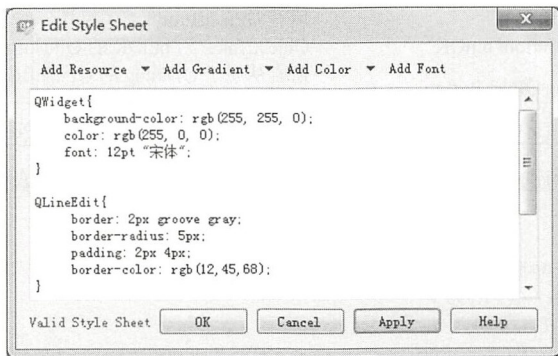


图 16-4 样式表编辑对话框

在图 16-4 中的对话框内，上方有几个具有下拉菜单的按钮，可以添加一些常用的样式属性，如前景色 color、背景色 background-color、选中后颜色 selection-color，以及背景图片 background-image 等。

## 16.2.2 Qt 样式表句法

### 1. 一般句法格式

Qt 样式表的句法（syntax）与 HTML 的 CSS 句法几乎完全相同。Qt 样式表包含一系列的样式法则，一个样式法则由一个选择器（selector）和一些声明（declaration）组成。例如：

```
QPlainTextEdit{
    font: 12pt "仿宋";
    color: rgb(255, 255, 0);
    background-color: rgb(0, 0, 0);
}
```

其中，QPlainTextEdit 就是选择器，表明后面花括号里的样式声明应用于 QPlainTextEdit 类及其子类。样式声明部分是样式法则列表，每个样式法则由属性和值组成，每条法则用分号结束。每条样式法则由“属性：值”构成，例如：

```
font: 12pt "仿宋";
```

表示 font 属性，字体大小为 12pt，字体名称为“仿宋”；当一个属性有多个值时，多个值用空格隔开。

2. 选择器（selector）

Qt 样式表支持 CSS2 中定义的所有选择器，表 16-1 显示的是一些常用的选择器。

表 16-1 Qt 样式表中的选择器类型

选择器	例子	用途
通用选择器	*	所有组件
类型选择器	QPushButton	所有 QPushButton 类及其子类的组件
属性选择器	QPushButton[flat=" false" ]	所有 flat 属性为 false 的 QPushButton 类及其子类的组件。如果样式表应用后组件的属性再发生变化，需要重新应用样式表才能刷新显示效果
非子类选择器	.QPushButton	所有 QPushButton 类的组件，但是不包括 QPushButton 的子类
ID 选择器	QPushButton#btnOK	ObjectName 为 btnOK 的 QPushButton 实例
从属对象选择器	QDialog QPushButton	所有从属于 QDialog 的 QPushButton 类的实例，即 QDialog 对话框里的所有 QPushButton
子对象选择器	QDialog > QPushButton	所有直接从属于 QDialog 的 QPushButton 类的实例

这些选择器的定义为选择界面组件提供了灵活性。选择器可以组合使用，一个样式声明可以应用于多个选择器，例如：

```
QPlainTextEdit, QLineEdit, QPushButton, QCheckBox{
    color: rgb(255, 255, 0);
    background-color: rgb(0, 0, 0);
}
```

这个样式声明将同时应用于 QPlainTextEdit、QLineEdit、QPushButton 和 QCheckBox 的实例。

```
QLineEdit[readOnly="true"], QCheckBox[checked="true"]
{ background-color: rgb(255, 0, 0) }
```

上面的这个样式应用于 readOnly 属性为 true 的 QLineEdit 和 checked 属性为 true 的 QCheckBox 实例，功能是使其背景颜色为红色。

在 Qt 中，可以为一个界面组件使用 QObject::setProperty() 设置一个动态属性，例如，在数据表编辑界面上，一些字段是必填字段，就可以为这些字段的关联组件设置一个 required 属性为 true，如：

```
editName->setProperty("required", "true");
comboSex-> setProperty("required", "true");
checkAgree-> setProperty("required", "true");
```

这样设置了三个界面组件的动态属性 `required` 为 `true`。

那么，可以应用下面的样式将这种必填字段的背景颜色设置为亮绿色。

```
*[required="true"] {background-color: lime}
```

### 3. 子控件 (sub-controls)

对于一些组合的界面组件，需要对其子控件进行选择，如 `QComboBox` 的下拉按钮，或 `QSpinBox` 的上、下按钮。通过选择器的子控件可以对这些界面元素进行显示效果控制。例如：

```
QComboBox::drop-down{ image: url(:/images/images/down.bmp); }
```

选择器 `QComboBox::drop-down` 选择了 `QComboBox` 的 `drop-down` 子控件，定义的样式是设置其 `image` 属性为资源文件中的图片 `down.bmp`。

```
QSpinBox::up-button{ image: url(:/images/images/up.bmp); }
```

```
QSpinBox::down-button{ image: url(:/images/images/down.bmp); }
```

这两条样式定义语句分别定义了 `QSpinBox` 的上、下按钮的图片，用资源文件中的图片替代了缺省的图片。

这样定义的 `QComboBox` 和 `QSpinBox` 具有如图 16-5 所示的显示效果。

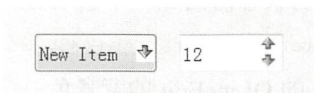


图 16-5 自定义了按钮图片的 `QComboBox` 和 `QSpinBox`

Qt 中常用的子控件见表 16-2，所有子控件的详细描述见 Qt 的帮助文档。

表 16-2 Qt 样式表中常见的子控件列表

子控件名称	说明
::branch	QTreeView 的分支指示器
::chunk	QProgressBar 的进度显示块
::close-button	QDockWidget 或 QTabBar 页面的关闭按钮
::down-arrow	QComboBox, QHeaderView (排序指示器), QScrollBar 或 QSpinBox 的下拉箭头
::down-button	QScrollBar 或 QSpinBox 的向下按钮
::drop-down	QComboBox 的下拉按钮
::float-button	QDockWidget 的浮动按钮
::groove	QSlider 的凹槽
::indicator	QAbstractItemView, QCheckBox, QRadioButton, 可勾选的 QMenu 菜单项, 或可勾选的 QGroupBox 的指示器
::handle	QScrollBar, QSplitter 或 QSlider 的滑块
::icon	QAbstractItemView 或 QMenu 的图标
::item	QAbstractItemView, QMenuBar, QMenu, 或 QStatusBar 的一个项
::left-arrow	QScrollBar 的向左箭头
::menu-arrow	具有下拉菜单的 QToolButton 的下拉箭头
::menu-button	QToolButton 的菜单按钮
::menu-indicator	QPushButton 的菜单指示器
::right-arrow	QMenu 或 QScrollBar 的右侧箭头
::pane	QTabWidget 的面板
::scroller	QMenu 或 QTabBar 的卷轴
::section	QHeaderView 的分段
::separator	QMenu 或 QMainWindow 的分隔器



续表

子控件名称	说明
::tab	QTabBar 或 QToolBox 的分页
::tab-bar	QTabWidget 的分页条。这个子控件只用于控制 QTabBar 在 QTabWidget 中的位置，定义分页的样式使用::tab 子控件
::text	QAbstractItemView 的文字
::title	QGroupBox 或 QDockWidget 的标题
::up-arrow	QHeaderView (排序指示器), QScrollBar 或 QSpinBox 的向上箭头
::up-button	QSpinBox 的向上按钮

4. 伪状态 (pseudo-states)

选择器可以包含伪状态，使得样式法则只能应用于界面组件的某个状态，也就是一种条件应用法则。伪状态出现在选择器的后面，用一个分号 (:) 隔开。如下面的样式法则：

```
QLineEdit:hover{
    background-color: black;
    color: yellow;
}
```

定义了当鼠标移动到 QLineEdit 上方时 (hover)，改变 QLineEdit 的背景色和前景色。

可以对伪状态取反，方法是在伪状态前面加一个感叹号 (!)，如：

```
QLineEdit:!read-only{ background-color: rgb(235, 255, 251); }
```

这定义了 readonly 属性为 false 的 QLineEdit 的背景色。

伪状态可以串联使用，相当于逻辑与的计算，例如：

```
QCheckBox:hover:checked{ color: red; }
```

这定义了当鼠标移动到一个被勾选了的 QCheckBox 组件上方时，其字体颜色变为红色。

伪状态可以并联使用，相当于逻辑或的计算，例如：

```
QCheckBox:hover, QCheckBox:checked{ color: red; }
```

这表示鼠标移动到 QCheckBox 组件上方，或 QCheckBox 组件被勾选时，字体颜色变为红色。

子控件也可以使用伪状态，如：

```
QCheckBox::indicator:checked{ image: url(/images/images/checked.bmp); }
QCheckBox::indicator:unchecked{ image: url(/images/images/unchecked.bmp); }
```

这里定义了 QCheckBox 的 indicator 在 checked 和 unchecked 两种状态下的显示图片，可以得到如图 16-6 所示的效果。



图 16-6 自定义图片作为 QCheckBox 的指示器

Qt 样式定义中常见的一些伪状态见表 16-3，熟悉这些伪状态并灵活应用可以定义自己想要的界面效果。

表 16-3 Qt 样式表中常见的伪状态

伪状态	描述
:active	当组件处于一个活动的窗体时，此状态为真
:adjoins-item	QTreeView::branch 与一个条目相邻时，此状态为真
:alternate	当 QAbstractItemView 的 alternatingRowColors() 属性为 true 时，绘制交替的行时此状态为真
:bottom	组件处于底部，如 QTabBar 的表头位于底部
:checked	组件被勾选，如 QAbstractButton 的 checked 属性为 true

续表

伪状态	描述
:closable	组件可以被关闭, 例如当 QDockWidget 的 DockWidgetClosable 属性为 true 时
:closed	条目处于关闭状态, 例如 QTreeView 的一个没有展开的条目
:default	条目是缺省的, 如一个缺省的 QPushButton 按钮, 或 QMenu 中一个缺省的 action
:disabled	条目被禁用
:editable	QComboBox 是可编辑的
:edit-focus	条目有编辑焦点
:enabled	条目被使能
:exclusive	条目是一个排他性组的一部分, 例如一个排他性 QActionGroup 的一个菜单项
:first	第一个项, 例如 QTabBar 中的第一个页
:flat	条目是 flat 的, 例如 QPushButton 的 flat 属性设置为 true 时
:focus	条目具有输入焦点
:has-children	条目有子条目, 例如 QTreeView 的一个节点具有子节点
:horizontal	条目具有水平方向
:hover	鼠标移动到条目上方时
:last	最后一个项, 例如 QTabBar 中的最后一页
:left	条目位于左侧, 例如 QTabBar 的页头位于左侧
:maximized	条目处于最大化, 例如最大化的 QMdiSubWindow 窗口
:minimized	条目处于最小化, 例如最小化的 QMdiSubWindow 窗口
:movable	条目是可移动的
:off	对于可以切换状态的条目, 其状态处于” off”
:on	对于可以切换状态的条目, 其状态处于” on”
:open	条目处于打开状态, 例如 QTreeView 的一个展开的条目
:pressed	条目上按下了鼠标
:read-only	条目是只读或不可编辑的
:right	条目位于右侧, 例如 QTabBar 的页头位于右侧
:selected	条目被选中, 例如 QTabBar 中一个被选中的页, 或 QMenu 中一个被选中的菜单项
:top	条目位于顶端, 例如 QTabBar 的页头位于顶端
:unchecked	条目处于未被选中状态
:vertical	条目处于垂直方向

## 5. 属性

Qt 样式表内对每一个选择器可定义多条样式规则, 每个规则是一个“属性: 值”对, Qt 样式表中可定义的属性很多, 可以在 Qt 的帮助文件中查找“Qt Style Sheets Reference”查看所有属性的详细说明。

在图 16-4 所示的样式表编辑对话框中, 从上方的几个按钮的下拉菜单中可以设计常用的一些属性, 如“Add Resource”按钮下三个菜单项可以从项目的资源文件中选择图片作为 background-image、border-image 或 image 属性的值; “Add Color”按钮的下拉菜单用于设置组件的各种颜色, 包括前景色、背景色、边框颜色等, 颜色的值可以用 rgb()、rgba() 函数表示, 或 Qt 能识别的颜色常量。

使用样式表可以定义组件复杂的显示效果。每个界面组件都可以用如图 16-7 所示的盒子模型 (Box Model) 来表示, 模型由四个同心矩形表示。

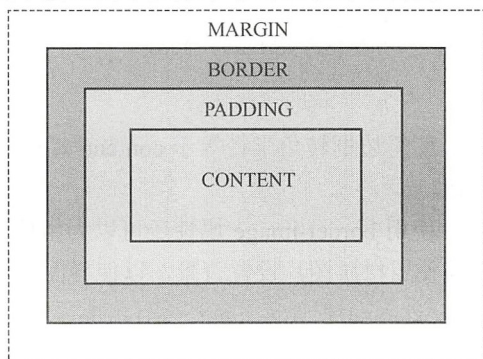


图 16-7 组件的盒子模型 (来自 Qt 帮助文件)

(1) `content` 是显示内容矩形区域, 如 `QLineEdit` 用于显示文字的区域, `min-width`、`max-width`、`min-height` 和 `max-height` 属性定义最大/最小宽度或高度, 就是定义这个矩形区, 例如:

```
QLineEdit{
    min-width:50px;
    max-height:40px;
}
```

这定义 `QLineEdit` 最小宽度为 50px, 最大高度是 40px, 其中 px 是单位, 表示像素。

(2) `padding` 是包围 `content` 的矩形区域, 通过 `padding` 属性可以定义 `padding` 的宽度, 或 `padding-top`、`padding-bottom`、`padding-left` 和 `padding-right` 分别定义 `padding` 的上、下、左、右宽度, 例如:

```
QLineEdit{ padding: 0px 10px 0px 10px;}
```

这设定 `padding` 的上、右、下、左的宽度, 它等效于:

```
QLineEdit{
    padding-top:0px;
    padding-right:10px;
    padding-bottom: 0px;
    padding-left:10px;
}
```

(3) `border` 是包围 `padding` 的边框, 通过 `border` 属性 (或 `border-width`、`border-style`、`border-color`) 可以定义边框的线宽、线型和颜色, 也可以分别定义 `border` 的上、下、左、右的线宽和颜色。使用 `border-radius` 可以定义边框转角的圆弧半径, 从而构造具有圆角矩形的编辑或按钮等组件, 例如:

```
QLineEdit{
    border-width: 2px;
    border-style: solid;
    border-color: gray;
    border-radius: 10px;
    padding: 0px 10px;
}
```

这使得 `QLineEdit` 具有灰色边框线条、圆角矩形的效果。

通过 `border-radius`、`min-width` 和 `min-height` 等属性可以设计圆形的按钮, 如:

```
QPushButton {
    border: 2px groove red;
    border-radius: 30px;
    min-width:60px;
    min-height:60px; }
```

使得边框转角半径等于 `content` 宽度或长度的一半, 宽度和长度相等, 就可以得到一个圆形的按钮。

使用 `border-image` 属性还可以为组件设置背景图片, 图片会填充 `border` 矩形框之内的区域, 一般使用材质图片设置背景, 以使界面具有统一的特色, 例如:

```
QLineEdit, QPushButton{border-image: url(:/images/images/border.jpg);}
```

(4) `margin` 是 `border` 之外与父组件之间的空白边距, 可以分别定义上、下、左、右的边距大小。



缺省的情况下, margin、border-width 和 padding 属性缺省值为零, 这种情况下, 四个同心矩形就是重合的一个矩形。

使用 Qt 样式表可以为界面组件设计各种美观的显示效果, 美观而特殊的界面不仅需要编程的能力, 更重要的是美工设计能力。

## 16.2.3 样式表的使用

### 1. 程序中使用样式表

有多种方法可以应用样式表。

第一种是在使用 Qt Designer 设计窗体时, 直接用样式表编辑器为窗体或窗体上的某个部件设计样式表, 则设计的样式保存在窗体的 ui 文件里, 窗体创建时会自动应用所设计的样式表。这样设计的样式表对应用程序是固定的, 无法取得换肤的效果, 而且需要为每个窗体都设计样式表, 重复性工作量大。

第二种是使用 setStyleSheet 函数应用样式, 使用 qApp 的 setStyleSheet() 函数可以为应用程序全局设置样式, 使用 QWidget::setStyleSheet() 可以为一个窗口、一个对话框或一个界面组件设置样式。例如:

```
qApp->setStyleSheet("QLineEdit { background-color: gray }");
```

这里使用应用程序全局变量 qApp 为 QLineEdit 设置样式, 如果应用程序内的某些 QLineEdit 组件没有再被设置样式, 则 QLineEdit 组件的背景色为灰色。

```
MainWindow->setStyleSheet("QLineEdit { background-color: lime }");
```

这是为主窗口 MainWindow 内的 QLineEdit 组件设置样式, 即背景色为亮绿色。

```
editName->setStyleSheet("color: blue;"
    "background-color: yellow;"
    "selection-color: yellow;"
    "selection-background-color: blue;");
```

这是设置一个 ObjectName 为 editName 的 QLineEdit 组件的样式, 注意这时在样式表中无需设置 selector 名称, 所设置的样式是应用于 editName 这个 QLineEdit 组件的。

这样将样式表固定在程序中, 很显然也是无法实现切换界面效果的。为了实现切换界面效果(换肤)的目的, 一般将样式定义表保存为 qss 后缀的纯文本文件, 然后在程序中打开文件, 读取文本内容, 再调用 setStyleSheet() 函数应用样式。示例代码如下:

```
QFile file(":/qss/mystyle.qss");
file.open(QFile::ReadOnly);
QString styleSheet = QString::fromLatin1(file.readAll());
qApp->setStyleSheet(styleSheet);
```

### 2. 样式定义的明确性

当多条样式法则对一个属性定义了不同值时, 就会出现冲突, 例如:

```
QPushButton#btnSave { color: gray }
QPushButton { color: red }
```



这两条法则都可以应用于 `ObjectName` 为 `btnSave` 的 `QPushButton` 组件，都定义了其前景色，这就会出现冲突。这时，选择器的明确性（specificity）决定组件适用的样式法则，即法则应用于更明确的组件。在上面的例子中，`QPushButton#btnSave` 被认为是比 `QPushButton` 更明确的选择器，因为它指向一个对象，而不是 `QPushButton` 的所有实例。所以，如果是在一个窗口上应用以上两条法则，则 `btnSave` 按钮的前景色为 `gray`，而其他按钮的前景色为 `red`。

同样，具有伪状态的选择器被认为比没有伪状态的选择器明确性更强，如：

```
QPushButton:hover { color: white }
QPushButton { color: red }
```

这样，当鼠标在按钮上停留时颜色为 `white`，否则颜色为 `red`。

如果两个选择器具有相同的明确性，则以法则出现的先后顺序为准，后出现的法则起作用，例如：

```
QPushButton:hover { color: white }
QPushButton:enabled { color: red }
```

这里的两个选择器具有相同的明确性，所以，当鼠标停留在一个使能的按钮上时，只有第二条法则起作用。这种情况下，如果希望不出现冲突，应该修改法则以使其更明确，如下面这两条法则是不冲突的。

```
QPushButton:hover:enabled { color: white }
QPushButton:enabled { color: red }
```

父子关系的两个类作为选择器时，具有相同的明确性，例如：

```
QPushButton { color: red }
QAbstractButton { color: gray }
```

这两个选择器的明确性相同，所以只依赖于语句的先后顺序。

确定法则的明确性，Qt 样式表遵循 CSS2 的规定，在设计样式表时应尽量明确并避免冲突情况。

### 3. 样式定义的级联性

样式定义可以在 `qApp`、窗口或一个具体组件中定义，任何一个组件的样式是其父组件、父窗口和 `qApp` 的样式的融合。当出现冲突时，组件会使用离自己最近的样式定义，即按顺序使用组件自己的样式、或父组件的样式定义、或父窗口的样式定义，或 `qApp` 的样式定义，而不考虑样式选择器的确定性。

例如，在 `QApplication` 中设置全局样式：

```
qApp->setStyleSheet("QPushButton { color: red }");
```

那么应用程序中所有未再定义样式的 `QPushButton` 的前景颜色为 `red`。

如果在 `MainWindow` 中再定义样式：

```
MainWindow->setStyleSheet("QPushButton { color: blue }");
```

则 `MainWindow` 上的按钮的前景色为 `blue`，而不是 `red`。

如果 `MainWindow` 上有一个名称为 `btnSave` 的 `QPushButton` 按钮，其定义样式如下：

```
btnSave->setStyleSheet("color: yellow; background-color: black;");
```

则按钮 btnSave 按照自己的样式显示前景和背景色。

Qt 样式表功能强大，可以设计自己想要的界面效果，但是这需要有较好的美工设计基础，需要在使用样式表的过程中不断尝试和总结经验。

## 16.3 使用 QStyle 设置界面外观

### 16.3.1 QStyle 的作用

Qt 是一个跨平台的类库，相同的界面组件在不同的操作系统上显示效果是不一样的。QStyle 是封装了 GUI 界面组件外观的抽象类，Qt 定义了 QStyle 类的一些子类，应用于不同的操作系统，如 QWindowsStyle 和 QMacStyle 等。这些样式是 Qt GUI 模块自带的，在不同的平台上编译运行的程序具有缺省的样式，QApplication::style() 可以返回应用程序缺省的样式。

Qt 内置的界面组件都使用 QStyle 进行绘制，以保证它们与运行平台的界面效果一致，如图 16-8 所示是 QComboBox 在不同操作系统上的九种不同的样式。

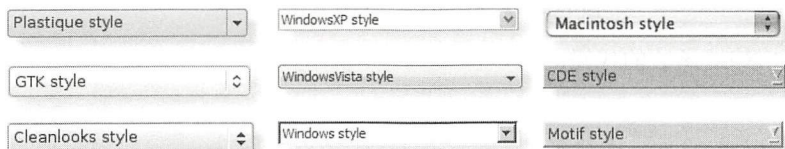


图 16-8 QComboBox 的九种不同样式（来自 Qt 帮助文件）

QStyleFactory 类管理着 Qt 的内置样式，QStyleFactory 有两个静态函数 keys() 和 create()，其函数原型如下：

```
QStringList QStyleFactory::keys()
QStyle *QStyleFactory::create(const QString &key)
```

keys() 函数返回一个字符串列表，是所在平台支持的 QStyle 的名称列表；create() 函数根据样式名称字符串创建一个 QStyle 对象。

QApplication 有两个静态函数用于操作样式，其函数原型为：

```
QStyle *QApplication::style()
void QApplication::setStyle(QStyle *style)
```

style() 函数返回应用程序当前的样式，任何一个 GUI 应用程序，在创建时就有缺省的样式，通过下面的语句：

```
QApplication::style()->metaObject()->className()
```

可以获得这个缺省样式的名称。

setStyle() 为应用程序设置一个样式，设置样式后，界面元素都具有这个样式所定义的外观。

除了这些 Qt 内置的样式，用户也可以从 QStyle 类继承，定义自己的样式，一般是从 QStyle 的子类 QProxyStyle 继承。

## 16.3.2 Qt 内置样式的使用

使用 Qt 内置的样式, 可以通过 `QStyleFactory::keys()` 获取运行平台支持的样式列表, 然后用 `QStyleFactory::create()` 创建样式, 再用 `QApplication::setStyle()` 设置样式即可。

创建一个基于 `QMainWindow` 的 Widget 应用程序 `samp16_2`, 并设计界面。如图 16-9 所示是设置为 `QWindowsStyle` 样式时的运行界面, 具有老式的 Windows 界面效果。



图 16-9 应用 `QWindowsStyle` 样式的界面

下面是主窗口构造函数的代码:

```
QMainWindow(parent),    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QRegExp regExp("(.*?)\\.+?Style");
    QString defaultStyle = QApplication::style()->metaObject()->className();
    ui->statusBar->showMessage(defaultStyle);
    if (regExp.exactMatch(defaultStyle))
        defaultStyle = regExp.cap(1);
    ui->comboSysStyle->addItem(QStyleFactory::keys());
    ui->comboSysStyle->setCurrentIndex(ui->comboSysStyle->findText(
        defaultStyle, Qt::MatchContains));
}
```

在构造函数中, 通过 `QStyleFactory::keys()` 获取当前平台上的样式名称列表, 并添加到下拉列表框 `comboSysStyle` 里。

程序运行后, 在“系统样式”下拉列表框里会显示当前平台支持的样式列表, 例如在 Windows 平台上, 列表里会有 `Widnows`、`WidnowsXP`、`WidnowsVista` 和 `Fusion`。在“系统样式”下拉列表框中选择一个样式可以为应用程序设置样式, 下拉列表框 `comboSysStyle` 的 `currentIndexChanged()` 信号响应槽函数代码如下:

```
void MainWindow::on_comboSysStyle_currentIndexChanged(const QString &arg1)
{
    // qApp->setStyle(arg1);
    QStyle *style=QStyleFactory::create(arg1);
    qApp->setStyle(style);
    ui->statusBar->showMessage(style->metaObject()->className());
}
```



这里首先使用 `QStyleFactory::create(arg1)` 根据选择的样式名称字符串 `arg1` 创建样式 `style`，然后使用 `qApp->setStyle(style)` 为应用程序设置样式。

如果不是需要显示样式的类名称，直接使用 `qApp->setStyle(arg1)` 也可以为应用程序设置样式。窗口上有“取消样式表”和“应用样式表”两个按钮，代码如下：

```
void MainWindow::on_btnNormal_clicked()
{ //取消所有样式表
    this->setStyleSheet("");
}
void MainWindow::on_btnStyleSheet_clicked()
{ //设置样式表
    this->setStyleSheet("QPlainTextEdit{color: blue; "
        "font: 13pt '宋体';}"
        "QPushButton:hover{background-color:lime;}"
        "QLineEdit{ border: 2px groove red;}"
        "background-color: rgb(170, 255, 127); "
        "border-radius: 6px;}"
        "QCheckBox:checked{color: red;}"
        "QRadioButton:checked{color:red;}"
    );
}
```

“应用样式表”按钮的代码为界面上的几个显示组件类设置了样式表，在设置样式表后，即使修改窗口样式，这些样式表定义的显示效果依然存在。

## 16.4 Qt 应用程序的发布

### 16.4.1 应用程序发布方式

用 Qt 开发一个应用程序后，将应用程序提供给用户在其他计算机上使用就是应用程序的发布。应用程序发布一般会提供一个安装程序，将应用程序的可执行文件及需要的运行库安装到用户计算机上，即使用户计算机上没有安装 Qt 也能正常运行安装的程序。

Qt 的应用程序发布有两种方式：静态链接和共享库方式。

静态链接（Static linking）是指用 Qt 编译应用程序时，将 Qt 的运行库等所需的支持文件全部静态编译到应用程序里，生成一个独立的可执行文件，应用程序发布只需很少的几个文件。这种方式的缺点是应用程序可执行文件很大，缺少灵活性。例如，当应用程序需要更新，或 Qt 有更新时，需要重新编译应用程序后再发布。而且，静态链接方式不能部署插件。

共享库（Shared Libraries）方式是指按正常方式编译生成应用程序，将应用程序运行所需的各种共享库与应用程序一同发布给用户。这样，当 Qt 的运行库更新时可以单独更新 Qt 运行库，应用程序如果使用了插件（插件是以共享库形式存在的），也可以单独更新插件，这为应用程序更新提供了方便。

如果要使用静态链接发布应用程序，还需要将 Qt 以静态方式重新编译生成静态版本的 Qt，然后用静态版本的 Qt 编译和链接应用程序，才可以生成静态链接的应用程序。这个过程显然很花



时间，也很复杂。所以，一般应用程序发布都采用共享库的形式。

## 16.4.2 Windows 平台上的应用程序发布

Qt 是跨平台的开发工具，开发的应用程序可以在各种平台上发布，本书只以常用的 Windows 平台为例介绍 Qt 应用程序的发布。

### 1. Windows 发布工具

windeployqt.exe 是 Qt 自带的 Windows 平台发布工具，它可以自动为一个应用程序复制其运行所需的各种库文件、插件和翻译文件，生成可发布的目录。

windeployqt.exe 文件在 Qt 的 bin 目录下，Qt 的每一个编译器版本均有独立的目录，如计算机上的 Qt 安装有 minGW32、MSVC2015 32bit 和 MSVC2015 62bit 3 个编译器版本，Qt 安装在 D:\Qt 目录下，则 3 个版本的 bin 目录是：

```
D:\Qt\Qt5.9.1\5.9.1\mingw53_32\bin
D:\Qt\Qt5.9.1\5.9.1\msvc2015\bin
D:\Qt\Qt5.9.1\5.9.1\msvc2015_64\bin
```

---

**注意** 应用程序由哪个编译器生成的，就应该用哪个版本的 windeployqt 生成发布文件。

---

在 Windows 的 Command 窗口使用 windeployqt 程序，其句法如下：

```
windeployqt [options] [files]
```

其中 options 是一些选项设置，可以查看 Qt 帮助文件查看具体设置，一般使用缺省设置即可，files 是需要生成发布文件的应用程序文件名。

要在 Command 窗口使用 windeployqt，最好的方法是将 windeployqt 版本所在的 bin 路径添加到系统的 PATH 环境变量里。

在 Windows 平台上，windeployqt 会将编译器的运行时文件复制到发布目录下，如果是用 MSVC 编译的，就包含 Visual C++ 的运行库。

### 2. MinGW 编译的应用程序发布实例

将 15.2 节的音频播放器实例程序 samp15\_1 用 MinGW 编译器在 release 模式下编译，生成可执行文件 samp15\_1.exe。将此可执行文件复制到一个目录下，例如“F:\Setup\minGW”，并将 samp15\_1.exe 更名为 AudioPlayer.exe。

将 MinGW 版本的 Qt 的 bin 路径添加到系统的 PATH 环境变量里，然后在 Windows 的“开始”菜单的文本框里键入“cmd”，回车后进入 Command 窗口，在 Command 窗口里依次键入并执行以下指令：

```
F:
cd setup\minGW
windeployqt AudioPlayer.exe
```

执行完成后，windeployqt 将 AudioPlayer.exe 运行时需要的各种库文件都复制到目录“F:\Setup\minGW”下，如图 16-10 所示。



图 16-10 MinGW 版本 AudioPlayer.exe 的发布文件

为测试生成的发布目录下的文件是否齐全，将环境变量 PATH 中的 Qt 的 bin 目录去除，然后双击图 16-10 中的 AudioPlayer.exe 文件运行程序。这时会出现一个对话框，提示丢失文件“libgcc\_s\_dw2-1.dll”，发现目录“D:\Qt\Qt5.9.0\5.9\mingw53\_32\bin”下有这个文件，将此文件复制到“F:\Setup\mingGW”目录下。

再次运行 AudioPlayer.exe 程序，还会出现丢失文件对话框，将这些文件复制到“F:\Setup\mingGW”下，依赖的库文件复制完全后，AudioPlayer.exe 才可以正常运行。

所以，windeployqt 并不能保证一次将所有依赖文件复制完全，需要测试运行应用程序以检验依赖库的完整性。

---

**注意** 运行应用程序，测试部署文件完整性时，必须删除系统环境变量里 Qt 的 bin 路径，否则系统从 Qt 的 bin 路径里总是可以找到依赖的库文件，就达不到测试的目的了。

---

查看一个可执行文件的依赖文件，也可以使用工具软件 Dependency Walker，它能可视化地查看可执行文件的依赖项。可以从其官网下载这个软件的最新版本。

即便在开发应用程序的计算机上测试发布程序没有问题，也应该将发布文件目录复制到一个没有安装 Qt 的计算机上测试应用程序能否正常运行。

### 3. MSVC 编译的应用程序发布实例

将 15.2 节的音频播放器实例程序 samp15\_1 用 MSVC2015 32bit 编译器在 release 模式下编译，生成可执行文件 samp15\_1.exe。将此可执行文件复制到一个目录下，如“F:\Setup\msvc32”，并将 samp15\_1.exe 更名为 AudioPlayer.exe。

然后就需要使用 MSVC2015 32bit 版本的 Qt 的 bin 目录下的 windeployqt 程序来为程序 AudioPlayer.exe 生成发布文件，其使用方法与 MinGW 版本的类似。生成的发布目录下的文件和目录与图 16-10 所示内容一样，只不过来源版本不同。

测试时双击 AudioPlayer.exe 文件，发现它可以直接运行，未提示丢失文件。

同样，也应该将发布目录复制到一个没有安装 Qt 和 MSVC2015 的计算机上测试，看其能否正常运行。

### 4. 安装程序制作

复制了应用程序所需的依赖文件，并测试没有问题后就可以制作安装文件了。有很多制作安装文件的软件，用户自行选择并下载一个即可制作自己的安装文件，这里不再详述。

# 欢迎来到异步社区！

## 异步社区的来历

异步社区 (www.epubit.com.cn) 是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD（按需印刷）结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



## 社区里都有什么？

### 购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

### 下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

### 与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

## 灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户账户中的积分可以用于购书优惠。100 积分 = 1 元，购买图书时，在  使用积分 里填入可使用的积分数值，即可扣减相应金额。



## 特别优惠

购买本书的读者专享异步社区购书优惠券。

使用方法：注册成为社区用户，在下单购书时输入 **S4XC5** 使用优惠券，然后点击“使用优惠码”，即可在原折扣基础上享受全单9折优惠。（订单满39元即可使用，本优惠券只可使用一次）

## 纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



## 社区里还可以做什么？

### 提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

### 写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在这一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

### 会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

## 加入异步

扫描任意二维码都能找到我们：



异步社区



微信服务号



微信订阅号



官方微博



QQ 群: 436746675

社区网址: [www.epubit.com.cn](http://www.epubit.com.cn)

投稿 & 咨询: [contact@epubit.com.cn](mailto:contact@epubit.com.cn)



# Qt 5.9

## C++ 开发指南

本书以 Qt 5.9 LTS 版本为开发平台,详细介绍了 Qt C++ 开发应用程序的技术,包括 Qt 应用程序的基本架构、信号与槽的工作机制、图形显示的 Graphics/View 架构、数据编辑和显示的 Model/View 架构、对话框和多窗口的设计与调用方法等。书中讲解了常用界面组件,文件读写,绘图、图表、数据可视化、数据库、多线程、网络和多媒体等模块的使用。每个编程主题均精心设计了完整的实例程序,所介绍的编程方法适用于 Qt 支持的任何平台。

本书适用于具有一定的 C/C++ 语言编程基础,想学习 Qt Creator 和 Qt 类库,并希望使用 Qt 开发各种应用程序的读者。

本书提供所有实例的源代码下载。

### 作者简介

#### 王维波

博士,主要从事地球物理探测仪器设计、数据处理方法研究和软件开发等工作,精通软硬件设计与开发。在实际研究和开发工作中发现 Qt 之利、Qt 之美,与读者分享 Qt 开发经验。

#### 栗宝鹃

博士,主要从事地球物理数据处理和成像方面的研究工作,在研究工作中将 Qt 用于专业软件的开发,精通 Qt 学习之道、应用之道。

#### 侯春望

硕士,主要从事单片机系统和应用软件的教学和研究工作,精通 Qt C++ 编程,曾开发多个专业应用软件。

 异步图书  
www.epubit.com



异步社区 [www.epubit.com](http://www.epubit.com)  
新浪微博 @人邮异步社区  
投稿/反馈邮箱 [contact@epubit.com.cn](mailto:contact@epubit.com.cn)

ISBN 978-7-115-47868-9



ISBN 978-7-115-47868-9

定价: 89.00 元

封面设计: 董志桢

分类建议: 计算机 / 程序设计 / C++

人民邮电出版社网址: [www.ptpress.com.cn](http://www.ptpress.com.cn)